NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

GENETIC ALGORITHM BASED OPTIMIZATION OF ADVANCED SOLAR CELL DESIGNS MODELED IN SILVACO ATLAS™

by

James Utsler

September 2006

Thesis Co-Advisors:	Sherif Michael
Thesis Co-Advisor	Bret Michael
Second Reader:	Todd Weatherford

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2006	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Genetic Algorithm Based Optimization of Advanced Solar Cell Designs Modeled in SIlvaco ATLAS™	5. FUNDING NUMBERS
6. AUTHOR(S) James D. Utsler	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

A genetic algorithm was used to optimize the power output of multi-junction solar cells. Solar cell operation was modeled using the Silvaco ATLAS™ software. The output of the ATLAS™ simulation runs served as the input to the genetic algorithm. The genetic algorithm was run as a diffusing computation on a network of eighteen dual processor nodes. Results showed that the genetic algorithm produced better power output optimizations when compared with the results obtained using the hill climbing/gradient approach.

14. SUBJECT TERMS Multi-junction Solar Cells, Distributed Computing, Genetic Algorithm, Optimization, Silvaco ATLAS™	15. NUMBER OF PAGES 109
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

GENETIC ALGORITHM BASED OPTIMIZATION OF ADVANCED SOLAR CELL DESIGNS MODELED IN SILVACO ATLAS™

James D. Utsler
Captain, United State Marine Corps
B.A. and B.S., University of Washington, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: James Utsler

Approved by: Sherif Michael
Co-Advisor

Bret Michael
Co-Advisor

Todd Weatherford
Second Reader

Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A genetic algorithm was used to optimize the power output of multi-junction solar cells. Solar cell operation was modeled using the Silvaco ATLAS™ software. The output of the ATLAS™ simulation runs served as the input to the genetic algorithm. The genetic algorithm was run as a diffusing computation on a network of eighteen dual processor nodes. Results showed that the genetic algorithm produced better power output optimizations when compared with the results obtained using the hill climbing/gradient approach.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1

II. BACKGROUND 3
A. SOLAR CELL BASICS (SEMICONDUCTOR BASICS) 3
1. Definition of a Material's Bandgap 3
2. The P-N Junction 8
3. Solar Cell Operation 10
4. Solar Cell Performance 11
a. Techniques of Characterization 11
b. Hindrances to Performance 13
B. MULTI-JUNCTION SOLAR CELL FUNDAMENTALS 14
1. Principle of Operation 14
2. Challenges with Multi-Junction Cells 16
a. Parasitic Junction and Tunnel Junctions 16
b. Materials Incompatibility 17
c. Shadowing Effect 18
d. Electrical Limitations 18
C. GENETIC ALGORITHMS 20
1. Concept and Typical Applications 20
2. Illustrative Example: Modeling the Single Junction Solar Cell 21
a. The Problem to be Solved 21
b. Numerically Representing the Problem 22
c. Executing the Algorithm 23
d. Breeding a New Generation 25
3. Nuances of Genetic Algorithms 28
a. Population Size 28
b. Selection Strategy 28
c. Crossover Settings 29
d. Probability of Mutation 30
D. MODELLING CELLS IN ATLAS™ SILVACO 31
1. Origins of the Cell Model 31
2. Previous NPS Research Progress 32
a. Michalopoulos 32
b. Green 32
c. Bates 33
d. Crespin 33

III. PREVIOUS OPTIMIZATION APPROACHES AND THE CASE FOR DISTRIBUTED COMPUTING 35
A. DREW BATES' GENETIC ALGORITHM AND ITERATIVE CURRENT MATCHING APPROACH 35
B. THE CASE FOR DISTRIBUTED COMPUTING 37

1. Size of the Solution Space 37
2. Distributed Computing Approach 38
3. Choosing a Distributed Computing Platform 39

IV. RESULT VALIDATION 43
A. DISTRIBUTED COMPUTING IMPLEMENTATION 43
1. Hardware Setup 43
2. Software Setup 43
a. Operating Systems 43
b. Distributed Computing Scheme 44
B. SINGLE-JUNCTION RESULT VALIDATION 45
1. Coarse Sampling and Gradient Ascent Method and Results 45
2. Distributed GA Method and Results 48
C. MULTI-JUNCTION CELL APPROACHES 50
1. GA Real-Values Method and Results 50
2. Possible Explanations for the Results 51

V. CONCLUSIONS 55

APPENDIX A DETAILED REVIEW OF AN INPUT DECK 57

APPENDIX B: CHALLENGES IN ADAPTING WINDOWS ATLAS™ INPUT DECKS TO ATLAS™ UNDER UNIX 65

APPENDIX C: DISTRIBUTED COMPUTING PROGRAMMER'S NOTES 69

APPENDIX D: MATLAB™ GENETIC ALGORITHM AND DIRECT SEARCH TOOLBOX NOTES 75

LIST OF REFERENCES 87

INITIAL DISTRIBUTION LIST 89

LIST OF FIGURES

Figure 1. Order of electron shell filling [From Ref. 3] 3
Figure 2. Silicon Electron Shell Diagram [From Ref. 4] 4
Figure 3. Silicon covalent bonds in a homogeneous mixture [After Ref. 5]..... 5
Figure 4. Energy band diagram for three types of materials [After Ref. 5]..... 6
Figure 5. N-Type doping using Arsenic (As) in Silicon [After Ref. 5]..... 7
Figure 6. P-type doping using Gallium (Ga) in Silicon [After Ref. 5] 8
Figure 7. Formation of the depletion region at (a) time zero and (b) equilibrium [After Ref. 5] 9
Figure 8. Electron-hole pair generation through a collision with a photon [After Ref. 5] 10
Figure 9. Solar cell in operation [After Ref. 5] 11
Figure 10. IV curve for a typical solar cell [After Ref. 6] 12
Figure 11. Irradiance plotted by wavelength (lower left) and contained energy (upper right) [After Ref. 6] 15
Figure 12. Simple stacking with parasitic junction(a) and tunnel junction (b) [From Ref. 6] 17
Figure 13. Typical Current-Voltage curves for solar cells based on various materials [From Ref. 6] 19
Figure 14. Example of Single Junction InGaP cell [After Ref. 7]..... 22
Figure 15. Encoding of traits into a 32-bit chromosome [From Ref. 7]..... 22
Figure 16. Simplified flowchart for genetic algorithm 24
Figure 17. Breeding Process Flowchart..... 25
Figure 18. Roulette wheel list generation (a) and selection mechanism (b) [From Ref. 7] 26
Figure 19. Dual-Point Crossover [From Ref. 7]..... 27
Figure 20. Mutation of Bit Values in a Chromosome [From Ref. 7]..... 27
Figure 21. Example of uniform crossover [From Ref. 8] 29
Figure 22. Results of Bates' iterative current matching routine for four-junction cell [From Ref. 7] 37
Figure 23. Original Lab Setup..... 43
Figure 24. Revised distributed computing platform..... 49
Figure 25. Highlighted Measurement Points for Individual Junction Layers 53
Figure 26. Comparison of IV characteristics obtained under Windows and Linux..... 67

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Quantization scheme for chromosome encoding [From Ref. 7]..... 23
Table 2. Comparison of Distributed Computing Approaches..... 40
Table 3. Computation Time required for varying granularity of coarse searches of the solution space..... 47
Table 4. Coarse Sampling and Gradient Ascent Data for InGaP Cells..... 47
Table 5. Extended Genetic Algorithm Results through 50 Generations with Increased Mutation Rate..... 50
Table 6. Sample Extraction code under Windows (Left) and Linux (Right) 66
Table 7. Scientific Constants used by ATLAS™ under Windows (Left) and Linux (Right)..... 67

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First, I would like to thank my thesis advisors, Professors Bret Michael and Sherif Michael. Their patient and tireless support throughout the research and writing process has kept me focused and on track to producing a useful body of work and to gain exposure through the writing and presentation of conference papers.

Second, I would like to thank Donna Burych for the countless hours spent assisting me with configuring the distributed computing system for ease of use and for timely troubleshooting and support throughout the process.

Next, I would like to thank Robin Jones at Silvaco. He provided timely responses to many of the day to day problems encountered using the solar cell models under Silvaco ATLAS^{TM}. Without him, I likely could not have deciphered some of Silvaco's error messages.

Last but not least, I would like to thank my wife, Tanya, and our children Isabelle, Timothy, and Casey, for their patience and understanding with the many hours of effort required to complete this research. After the first eight years of our marriage involved in a distance MBA and this resident MSEE program, we will finally have evenings without academic studies. I thank Tanya especially for bearing a disproportionate share of the work to maintain our home and raise our children during this time. My success is as much a product of her effort as mine.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The current state of the solar cell industry is that development occurs by the fabricate-and-test method. Engineers design cells, have them fabricated, test them in the lab, and then repeat the process. The industry has not found any suitable software platforms which may be used to accurately predict or simulate cell performance. Over the past four years, researchers at NPS have found a way to model solar cells within Silvaco's ATLAS™ software. ATLAS™ is a physically based simulator which models the flow of charge carriers through a device based on the physical structure defined by the user.

Solar cell models for single, dual, triple, and quad-junction solar cells have been modeled. The single, dual, and triple-junction cell models have been validated against experimental results for accuracy. The quad-junction cell is a design proposal and has not yet been fabricated. ATLAS™ cell models have been used for validation against experimental results and to optimize designs for future cells. In addition, radiation effects on a single-junction cell have been successfully modeled and validated against experimental data.

The optimization of cell designs was initially a trial and error process. One researcher, Drew Bates, used a genetic algorithm to optimize the individual layers of a multi-junction cell and developed an iterative current-matching routine for the optimization of the combined multi-junction cell. Bates admitted that his optimization was limited by the amount of computation time available during his time as a student. This thesis explores the validity of Bates' approach by using a coarse sampling and gradient ascent algorithm as well as a variant of the genetic algorithm to more completely explore the solution space. To accommodate a larger sample of the solution space, a distributed computing platform was developed and implemented.

Bates genetic algorithm implementation focused on seven cell traits modeled with 16 possible values for each trait. For each cell type tested, a

coarse sampling of the solution space was conducted by taking all permutations of three values of each trait: the low, middle, and high values. This produced 2187 unique chromosomes spread evenly throughout the solution space. Following simulation, the five best candidates were subjected to a gradient ascent search. The gradient process simulated a list of chromosomes derived by taking every permutation of trait values equal to, one higher, and one less than the candidate's trait values. Once simulations were complete, the next gradient search was centered on whatever point had the maximum power output. This gradient ascent method continued until a local maximum was reached. After exercising the algorithm on each cell type, no improvements from Bates' results were obtained. This confirms the nonlinearity of the solution space for this problem. If each of the seven traits had a linear effect on output power, the gradient search method would always lead to the global maximum. In most searches, each candidate followed its gradient to a different local maximum.

The second approach was to apply a genetic algorithm using the MATLAB™ Genetic Algorithm and Direct Search Toolbox as a front-end combined with the distributed computing system as the back end. Genetic algorithm properties used by Bates were preserved with two exceptions. The first is that the simulations were continued out to 50 generations vice 20. The second exception was that the mutation rate was increased because the populations were observed to converge by the 20 generation mark. In almost every cell configuration tested, an improvement in cell output power was obtained.

Bates utilized an iterative current-matching routine to optimize multi-junction cells. Quad-junction cells are essentially four separate solar cells stacked on top of one another and wired in series. While the overall voltage is the sum of the four layers, the current is limited by the layer producing the least current. The current-matching routine works by adjusting cell thicknesses to match the currents. For example, the top layer normally produces the highest voltage and lowest current. If the top layer is made thicker, its voltage remains

unchanged while its current production increases. In addition, more light is absorbed and less is transmitted through to the other layers. This reduces current production in the other layers but brings the total cell to a higher output power by increasing the top layer's limiting current level. As this process was used, many cases occurred where layer thicknesses were needed which had not been specifically optimized for using the genetic algorithm. In these cases, approximations were made using thickness and power values of the nearest known cell configurations.

In the final experiment of this thesis, a real-valued genetic algorithm was applied to the total quad-junction cell. The results of this algorithm were inconclusive. In each generation of results, the power values for an entire population would have an identical power value even though the cell configurations were different. When known optimized cell parameters were inserted into the population, those power values would match previous results while the remainder of the population would have a common power value different than the optimized cell. The root cause of this needs to be investigated further.

In initial investigation, two sources of error were found. The first is in the way multi-junction cells are modeled in ATLAS™. At this time, there is not a working model of the tunnel junction between the layers of the cell. To model the cell, the space occupied by the tunnel junction is modeled by a vacuum with optical properties that don't cause any refraction between the two cell layers. Separate contacts for each layer are modeled and the IV curve is extracted for each of the individual layers. This ties into the second source of error: the IV curve solve points found by ATLAS™ are focused around each junction layer's maximum power point since this has been the focus of research. However, when three other junction layers are limited to the fourth layer's current, in most cases there are no IV curve solve points for the other three layers at that current level since it's not near the layers' maximum power current. When this occurs, the MATLAB™ algorithm which computes output power conducts a linear

approximation based on the two nearest points. While this sometimes falls on a very linear portion of the IV curve, it sometimes does not.

The development of a tunnel junction for the multi-junction cell model is the most promising remedy to this problem. It would allow the direct measurement of the cell's total output power without the need to measure each junction layer independently and perform calculations.

Through this research, Drew Bates' optimization approach for single-junction cells using the genetic algorithm was validated. A coarse sampling and gradient ascent algorithm did not find improved solar cell output power values. Improved results were found in this research only by continuing his algorithm for more generations and with a higher mutation rate. The optimization of the quad-junction in this thesis was inconclusive, The cell itself and optimization approaches can be significantly improved by the development of the tunnel junction within the ATLAS^{TM} multi-junction cell model.

I. INTRODUCTION

In 1839, Antoine-Cesar Becquerel discovered that some combinations of materials produced electricity when exposed to light [Ref: 1]. The first cell with similar construction to modern cells was fabricated by Charles Fritts in 1877 by coating selenium with a nearly transparent thin layer of gold [Ref: 1]. However, his cells were less than 1 percent efficient in converting the received light energy into useful electric current [Ref: 1]. While minor improvements were made up through the 1930's, solar cells were not considered as a potential power source until Russell Ohl developed the first silicon solar cell in 1941 [Ref:1]. Subsequent improvements by Pearson, Chapin, and Fuller brought the cell's efficiency up to 6 percent in 1954 [Ref. 1].

Today's single-junction solar cells range in the 15-20 percent range. Triple-junction solar cells, with individual junctions stacked on top of one another, have been fabricated with an advertised efficiency of 29.3 percent [Ref. 2]. The field is always expanding with research on various construction techniques and potential new compounds for use in the cells. Recently proposed designs could increase cell efficiency well into the mid 30 percent range. However, at this time, these types of cells are too costly for most applications.

Unlike terrestrial applications, spacecraft operate outside the light-degrading effects of Earth's atmosphere; solar cells are exposed to significantly more solar energy. In addition, solar cells offer one of the only renewable energy sources for a satellite in orbit. With the cost of putting something in orbit around the Earth in the vicinity of $10,000 per pound, acquiring more advanced high-cost, high-efficiency, multi-junction designs can be justified.

While the industry standard solar cell development process is fabricate-and-test, researchers at the Naval Postgraduate School have developed a software based model of solar cells which closely replicates the performance of well-documented experimental cells of the same design. More recent efforts

have turned to using the computer model as a design tool to optimize certain cell parameters in order to attain maximum power output. The most recent optimization approach used a genetic algorithm to improve solar cell performance. In this thesis, the results Bates obtained through experimentation with the genetic algorithm were compared with results obtained using the coarse sampling/gradient ascent approach as well as a modified genetic algorithm approach.

Chapter II covers solar cell operation, the theory of genetic algorithms, and modeling solar cells using the Silvaco ATLAS^{TM} software. Chapter III describes the previous optimization approach studied in this thesis. Chapter IV gives the approach used for this thesis and the results obtained. Chapter V gives conclusions and recommendations for further extension of this work. Appendix A gives a more detailed description of the input decks used for Silvaco's ATLAS^{TM} software. Appendix B discusses the challenges in adapting input decks from the Silvaco software running under Windows to a Linux-based computer. Appendix C gives programmer's notes and code excerpts on how the distributed computation was accomplished. Appendix D gives programmer's notes on challenges faced using the MATLAB^{TM} Genetic Algorithm and Direct Search Toolbox.

II. BACKGROUND

A. SOLAR CELL BASICS (SEMICONDUCTOR BASICS)

Solar cells are essentially electronic devices that convert energy received from a light source into usable electricity. Their construction and operation is based on several material properties and some unique behavior when the materials are combined in a specific way. This section documents those properties and combinations which allow a single junction solar cell to operate.

1. Definition of a Material's Bandgap

At present, silicon (Si) is the basis of construction of the majority of solar cells. The reason for this has to do with Si's unique atomic structure and material properties. Recall that Si is a Group IV element with atomic number 14. The Group IV designation denotes four electrons in its outermost shell. This occurs because, as the atomic number increases, electron shells are filled in the following order:

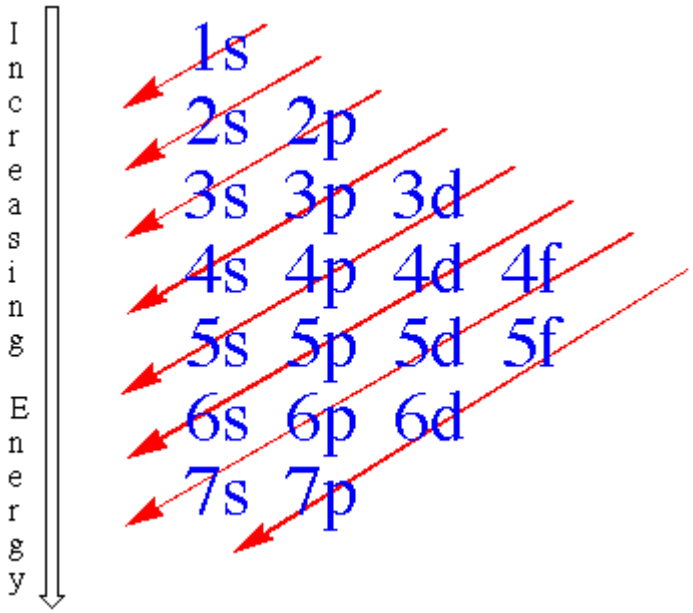


Figure 1. Order of electron shell filling [From Ref. 3]

For Si, this means that in the first energy level, two electrons occupy the 1s orbital. Within the second energy level, two electrons occupy the s orbital and six occupy the p orbital. Finally, within the third energy level, two electrons occupy the s orbital and two occupy the p orbital. This makes a total of 14 electrons.

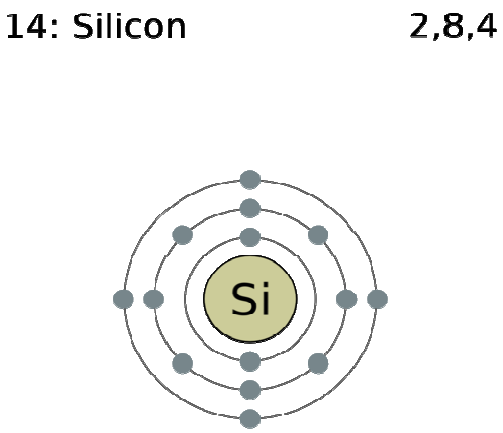


Figure 2. Silicon Electron Shell Diagram [From Ref. 4]

As a general rule, most elements are more stable when they contain a total of eight electrons in their outer, or valence, shell. Si normally accomplishes this through the formation of covalent bonds with other Si atoms. In this way, a Si atom surrounded by four other Si atoms can share one electron with each of the other four in order to have a “complete” outer shell. Note that in the following diagram, only the outer shell electrons are shown.

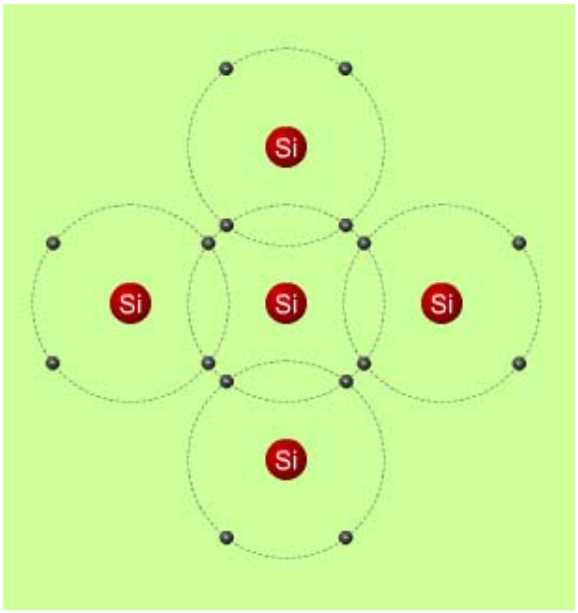


Figure 3. Silicon covalent bonds in a homogeneous mixture [After Ref. 5]

The center Si molecule shown above has eight electrons occupying its outermost shell. The atom is fairly stable in this configuration and the eight electrons are said to be in the valence band: these electrons remain with the Si atom, requiring an external influence to break an electron free. A free electron derived from a donor atom is said to occupy the conduction band. In the conduction band, the electron is free to move throughout the material. When an electron leaves its donor atom, a “hole” with net positive charge is created. The energy required to promote an electron from the valence band to the conduction band is called the material’s bandgap, as shown in Figure 5.

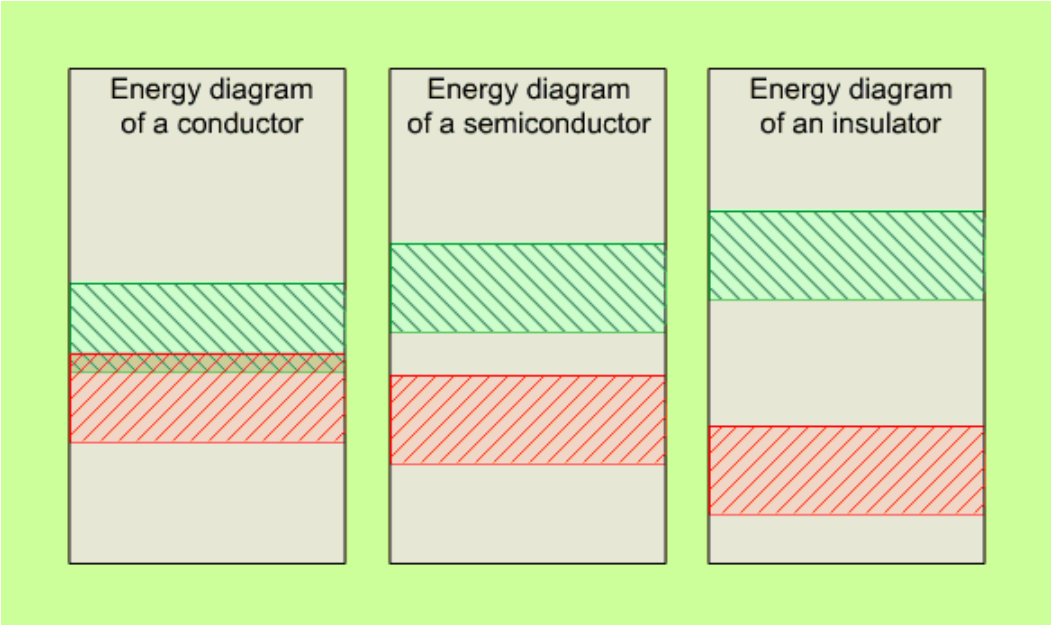
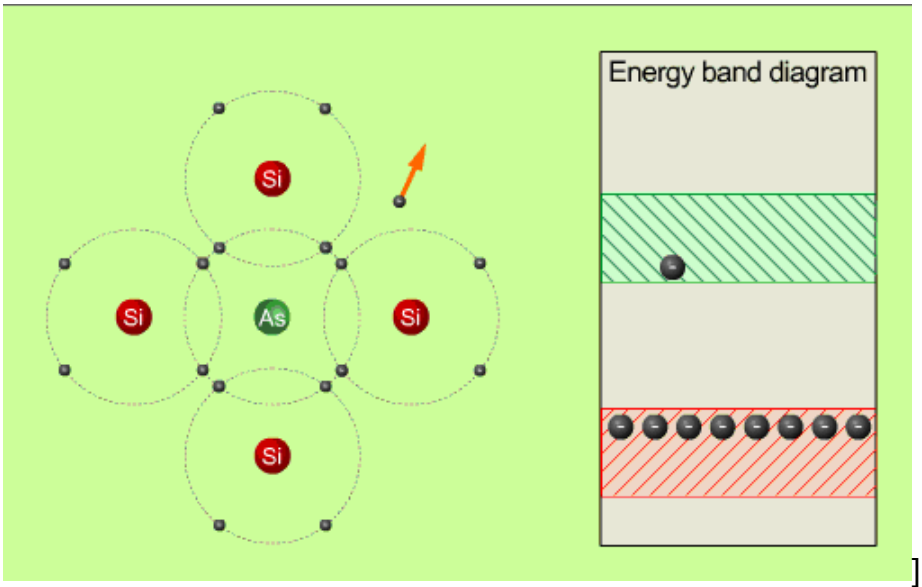


Figure 4. Energy band diagram for three types of materials [After Ref. 5]

The green region represents the conduction band, the red region represents the valence band, and the space in between is the bandgap. No energy is required to allow electrons to move about freely within a conductor. Conversely, a significant amount of energy is required to allow an insulator's electrons to move about freely. Semiconductors, of which Si is classified, fall somewhere in the middle. At zero Kelvin, all electrons are in the valence band. Once above zero Kelvin, electrons can gain enough energy introduced through temperature or other external influences to move into the conduction band. However, the number of electron-hole pairs formed in homogeneous Si at room temperature is on the order 10^{10} total electron-hole pairs in a cubic centimeter of Si [Ref. 6].

In order to increase the number of electron-hole pairs in a given volume, a doping process may be used. Doping is a process by which impurities are added to Si when it is made. For solar cell applications, doping is typically conducted with Group III or Group V elements. For our first example, consider the addition of arsenic (As) to pure Si. Arsenic is a Group V element with five electrons in its outer shell. When placed in Si, it forms covalent bonds with four surrounding Si

atoms, creating an outer shell with nine electrons, four shared and five in As's outer shell. The energy required for the ninth electron to reach the conduction band is extremely low. As shown in the energy band diagram of Fig. 6, the ninth electron moves into the conduction band. This is called n-type doping since it frees negative charge carriers to move within the material.



]

Figure 5. N-Type doping using Arsenic (As) in Silicon [After Ref. 5]

P-type doping, on the other hand, introduces a Group III material into the Si. In this example, gallium (Ga) is added. Since Ga only has three electrons in its outer shell, covalent bonds with four adjacent Si atoms leave Ga with only seven electrons in its outer shell. The missing electron is referred to as a hole. The hole is a positive charge carrier and is the basis for calling this a p-type material.

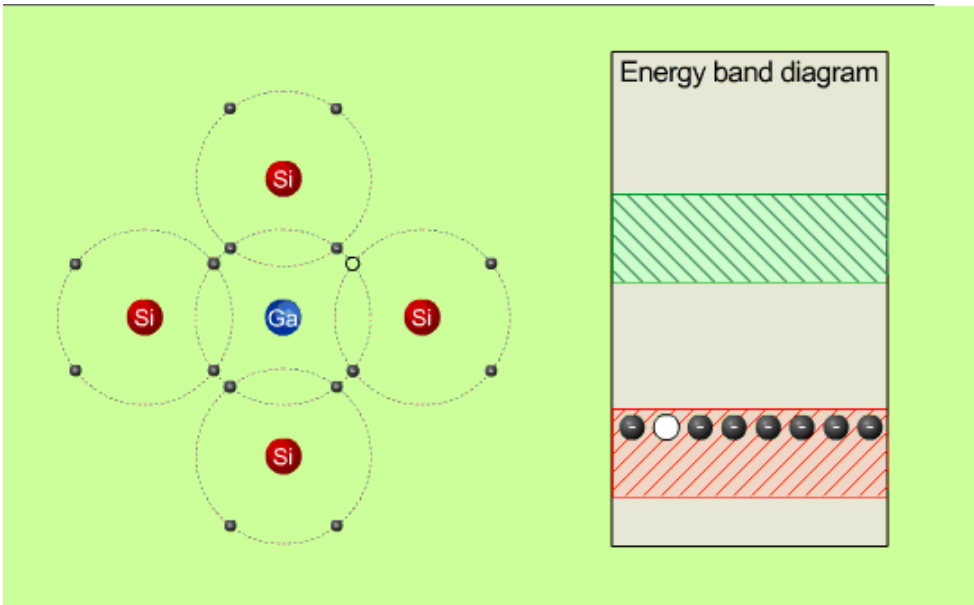


Figure 6. P-type doping using Gallium (Ga) in Silicon [After Ref. 5]

2. The P-N Junction

Recall that even though a n-type material has electrons in the conduction band, it is electrically neutral since the total material has just as many protons as electrons. In addition to electrons, which are the majority carrier, a small number of electron hole pairs are formed at room temperature simply by thermal energy. The holes created from this process are called the minority carrier for a n-type material. Consider what would happen when a thin layer of p-type material is placed in direct contact with an n-type material. The first event that occurs is that a small number of free electrons from the n-type material near the junction move to fill holes in the p-type material. The movement of electrons out of the n-type material leaves it positively charged. Conversely, the addition of electrons to the p-type material gives it a negative charge. Similarly, a small number of holes move from the p-type material across the junction into the n-type material where they combine with electrons. This leaves a region of the n-type material with a positive charge and a region of the p-type with a negative charge as with the movement of electrons. This region is known as the depletion zone of a P-N junction and the movement of carriers is known as diffusion current. This effect quickly reaches an equilibrium because of a resulting electro-static field. The

slightly positive and negative “poles” of the depletion region create a field pointing from the n-type to the p-type material (from positive to negative charge). Since like charges repel, the negative field in the depletion region of the p-type material prevents movement of further electrons from the n-type material to the depletion zone. The same effect occurs to prevent further movement of holes from the p-type material into the depletion region. Recall that in the n-type material there are a small number of naturally occurring holes termed minority carriers. This electro-static field of the depletion region sweeps all minority carriers into the depletion region.

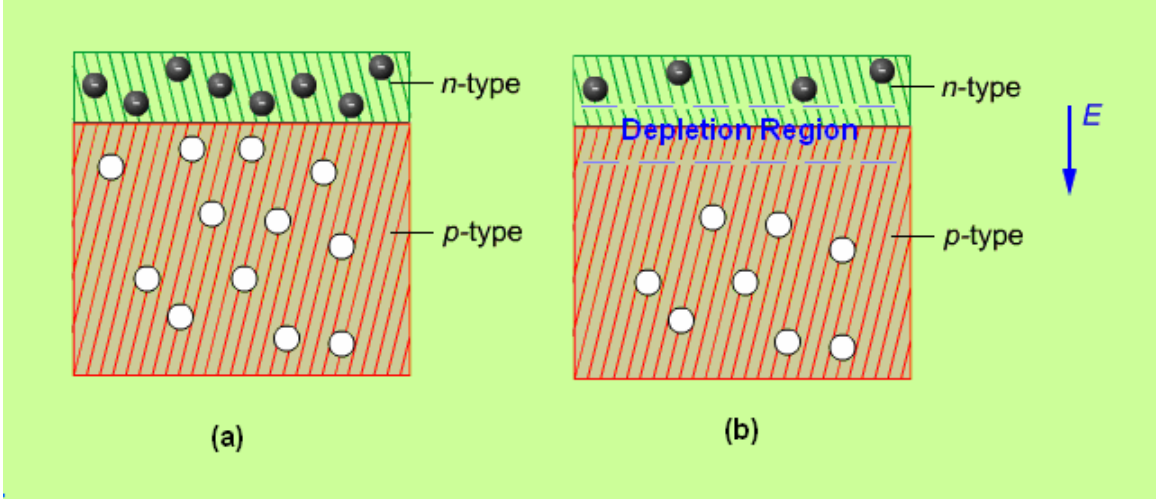


Figure 7. Formation of the depletion region at (a) time zero and (b) equilibrium [After Ref. 5]

At first inspection, it seems that one could attach contacts to the top and bottom of this material and have a limitless source of energy. However, junction effects between the semiconductor materials and the contact conductors prevent the junction alone from generating energy. However, if an external energy source can supply energy to the P-N junction in order to create electron-hole pairs, a useful current may be created.

3. Solar Cell Operation

A solar cell operates through the introduction of energy into the P-N junction. Photons are the fundamental particle of energy transmission using light. Photons traveling at the speed of light contain energy. When a photon travels into a P-N junction, it can be absorbed by the material in the junction to create an electron-hole pair.

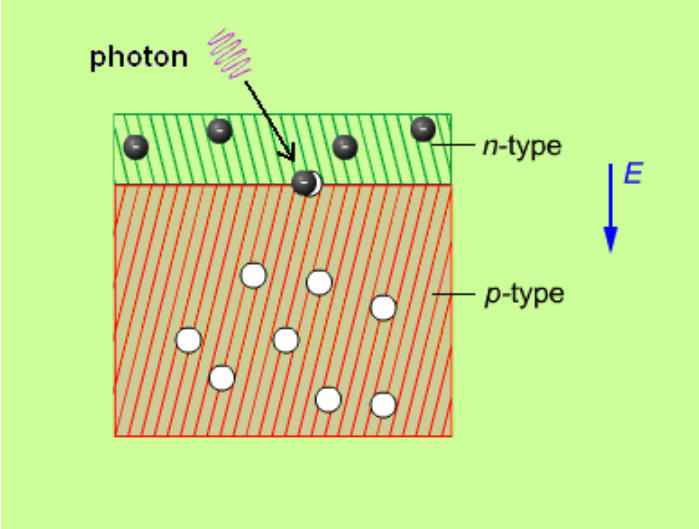


Figure 8. Electron-hole pair generation through a collision with a photon [After Ref. 5]

Without a conductor on the top and bottom of the P-N junction, the electron and hole would move around and eventually recombine to release energy in the form of heat. However, if the P-N junction is made sufficiently thin, an electron generated in the n-type material will be swept into the top contact and its corresponding hole will be swept into the depletion region because of the electro-static field. The electron will then travel through a circuit and then recombine with a hole generated in the p-type material or be swept back into the depletion region by the electro-static field. Conversely, holes created in the p-type material move in the opposite direction: this creates a current in the circuit which may be harnessed to accomplish useful work.

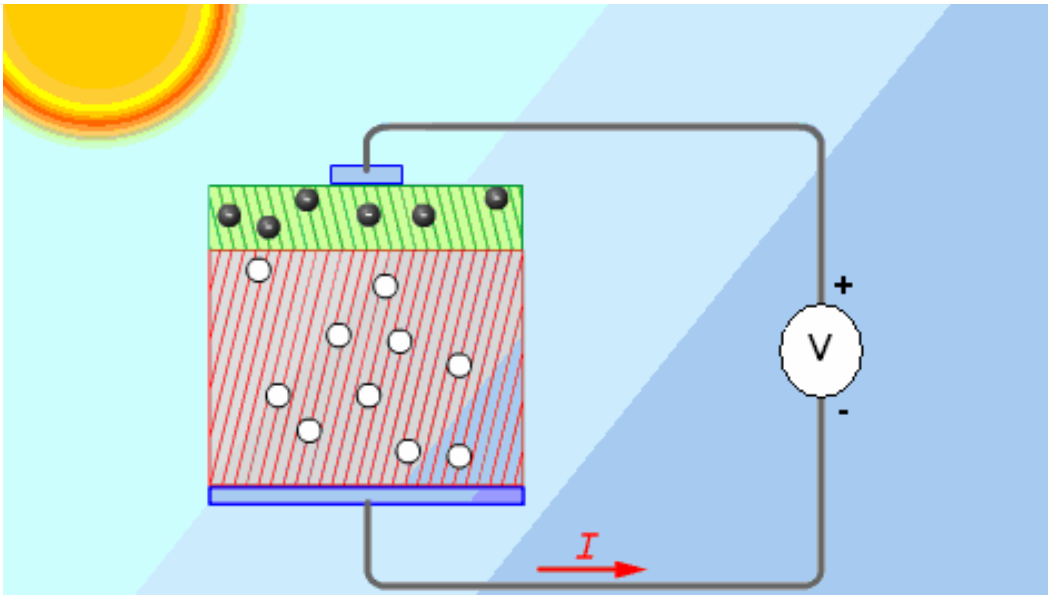


Figure 9. Solar cell in operation [After Ref. 5]

4. Solar Cell Performance

a. Techniques of Characterization

In measuring solar cell performance, standard electrical units are used. Typical benchmarking of cells occurs using specially designed lighting equipment which accurately reproduces the spectral content and intensity of light encountered in space. The normal means of displaying this data is through the current-voltage (IV) curve of the device. The curve indicates what voltage and current the device will produce for a given load. In addition, it defines the open-circuit voltage (V_{oc}) and short-circuit current (I_{sc}) for the cell. The V_{oc} is the maximum voltage the device will produce and is measured with no load attached to the device. The I_{sc}, on the other hand, is the maximum current the cell can produce and is measured with a zero voltage or short circuit. Figure 10 shows the elements described above.

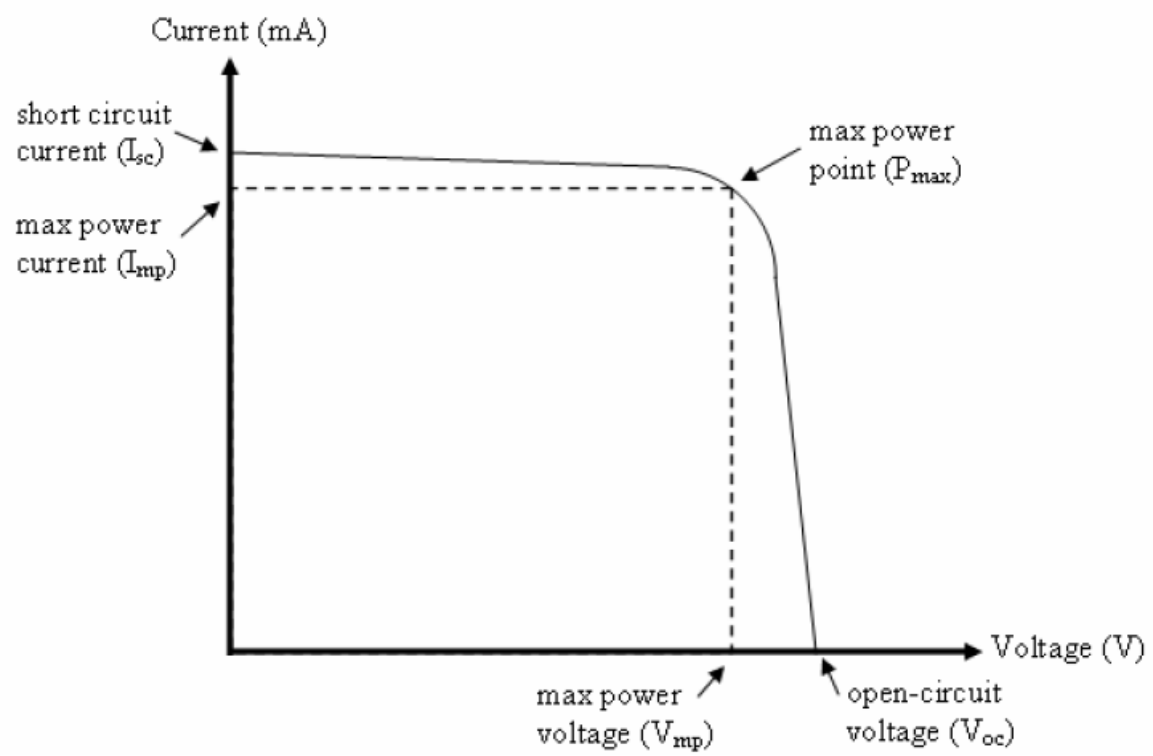


Figure 10. IV curve for a typical solar cell [After Ref. 6]

Based on the IV curve of a solar cell, a couple benchmarks for cell performance can be derived. The first, efficiency, is based on how much output power is generated compared to the amount available from the light source [Ref. 6].

$$\eta = \frac{P_{mp}}{P_{in}} \bullet 100\% \tag{1}$$

In equation (1), η is the efficiency of the cell, P_{in} is the power provided by incoming light, and P_{mp} is the power generated by the cell calculated using equation (2).

$$P = I \bullet V \tag{2}$$

Second, the fill factor is a measure of the sharpness of the knee of the IV curve. A fill factor of 1% would be a flat curve while a fill factor of would be a right angle. Equation (3) shows how the fill factor is derived [Ref 6].

$$FF = \frac{P_{mp}}{V_{oc} \bullet I_{sc}} \bullet 100\% \tag{3}$$

Note that efficiency and fill factor may be derived from data on the IV curve as long as the incident light intensity is known. For this thesis, efficiency calculations are based on an incident light energy of 135 milliwatts per square centimeter. Current single junction solar cells have efficiencies in the 15% range.

b. Hindrances to Performance

The factors affecting cell performance are numerous, such as those listed below from Ref.5:

1. Light incident on a cell's surface is prone to reflection. This is a combination of the angle of incidence of the light and material properties. An angle of incidence far from perpendicular combined with a highly reflective material on the cell's surface may account for up to 36% reflection of the incoming photons. Specially designed anti-reflective coatings on a cell's surface may reduce the amount of light reflected to approximately 5% as long as the angle of incidence is close to perpendicular.

2. Not all photons are created equal. Some photons do not have sufficient energy to promote an electron from the valence band to the conduction band. However, these photons can still be absorbed and result in the generation of heat. Heat in an electrical device yields increased resistance and a lowering of cell performance.

3. Photons with too much energy will promote an electron to the conduction band and also generate excess heat.

4. While the electro-static field of the depletion region sweeps charge carriers to opposite sides of the cell, some internal recombination does occur with a resulting heat gain.

5. Resistance in the metal contact materials causes a drop in output power and increases cell temperature.

6. While the manufacture of solar cells is a refined process, it is still subject to material defects. Imperfections in the semiconductor crystal structures degrade cell performance.

7. The conducting grid on the top of a cell shades approximately 8% of its top surface area. These contacts do not allow light to pass through into the cell.

8. If the cell is above or below its designed operating temperature, the vibration of the crystal lattice structure will interfere with the movement of charge carriers through the cell.

9. A photon is a very small particle as are the atoms in a crystal structure. Not all photons traveling into a solar cell will be absorbed by a semiconductor atom. Some of this effect is mitigated through the addition of a reflective surface on the bottom of the cell. This doubles the opportunity for absorption by forcing the photon back through the cell on its return trip.

B. MULTI-JUNCTION SOLAR CELL FUNDAMENTALS

1. Principle of Operation

As discussed in part A, a material's bandgap defines the amount of energy required to move an electron from the valence band to the conduction band. Light photons contain varying amounts of energy. The energy contained is inversely related to the wavelength of light which contains the energy. The equation

$$E = \frac{hc}{\lambda} \tag{4}$$

defines the energy E in Joules where h is Planck's constant (4.136×10^{-15} eV·sec), c is the speed of light (3.0×10^{8}m/sec), and λ is the wavelength of light being considered [Ref. 6]. In total, there is approximately 130 milli-Watts(mW) per square centimeter (cm^2) of energy available in Earth orbit [Ref. 6]. Since h and c are constants, the equation may be reduced to:

$$E = \frac{1.24}{\lambda} \tag{5}$$

where λ is measured in micrometers (microns) and E is in electron volts. Light conditions in Earth orbit are commonly referred to as Air Mass Zero (AM0). The light energy, however, is spread out among various wavelengths.

A solar cell may be tuned to respond to different parts of the light spectrum by adjusting the materials and construction of a cell. However, there is not a single cell material which absorbs the entire spectrum of light. The following figure shows the amount of energy contained in light in Earth orbit (AM0) according to the wavelength of light and energy contained. The spectral response of three cell types are plotted too. The plots show which portions of the spectrum each of the different cell materials can capture to produce electricity. The purple curve represents the total irradiance at given wavelength or energy level in AM0.

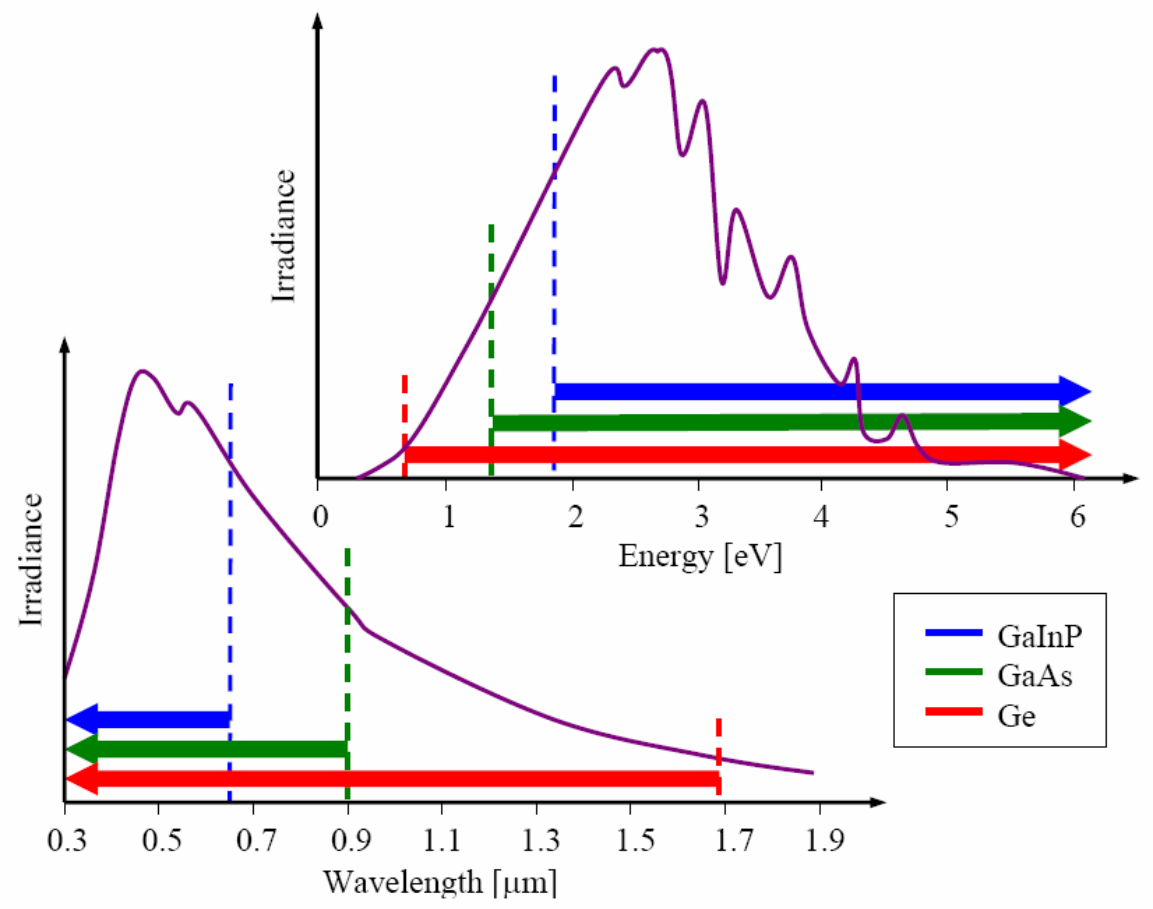


Figure 11. Irradiance plotted by wavelength (lower left) and contained energy (upper right) [After Ref. 6]

2. Challenges with Multi-Junction Cells

a. Parasitic Junction and Tunnel Junctions

The goal in designing multi-junction cells is to select enough layers with varying properties in order to capture and efficiently convert as much of the available light spectrum as possible. The Holy Grail of solar cells is to reach 100% efficiency and generate 130mW/cm^2 for every solar cell on a spacecraft. However, all the conditions mentioned at the beginning of this chapter which degrade solar cell performance also apply to multi-junction solar cells. In addition, there is one technical difficulty with “stacking” individual junctions on top of one another in a single cell. When two P-N junctions are put in direct contact with one another, a parasitic P-N junction is formed between them with an electro-static field opposing the flow of current between the two junctions as shown in Figure 12a. This parasitic junction is strong enough to cause unacceptable electrical losses within the cell by opposing current movement. To mitigate the electrical losses, the introduction of a heavily doped reverse-biased P-N junction between the two cells allows current to flow with minimal loss. This P-N junction is called a tunnel junction and it creates an electro-static field in the same direction as the P-N junctions of the top and bottom junction layers per figure 12b.

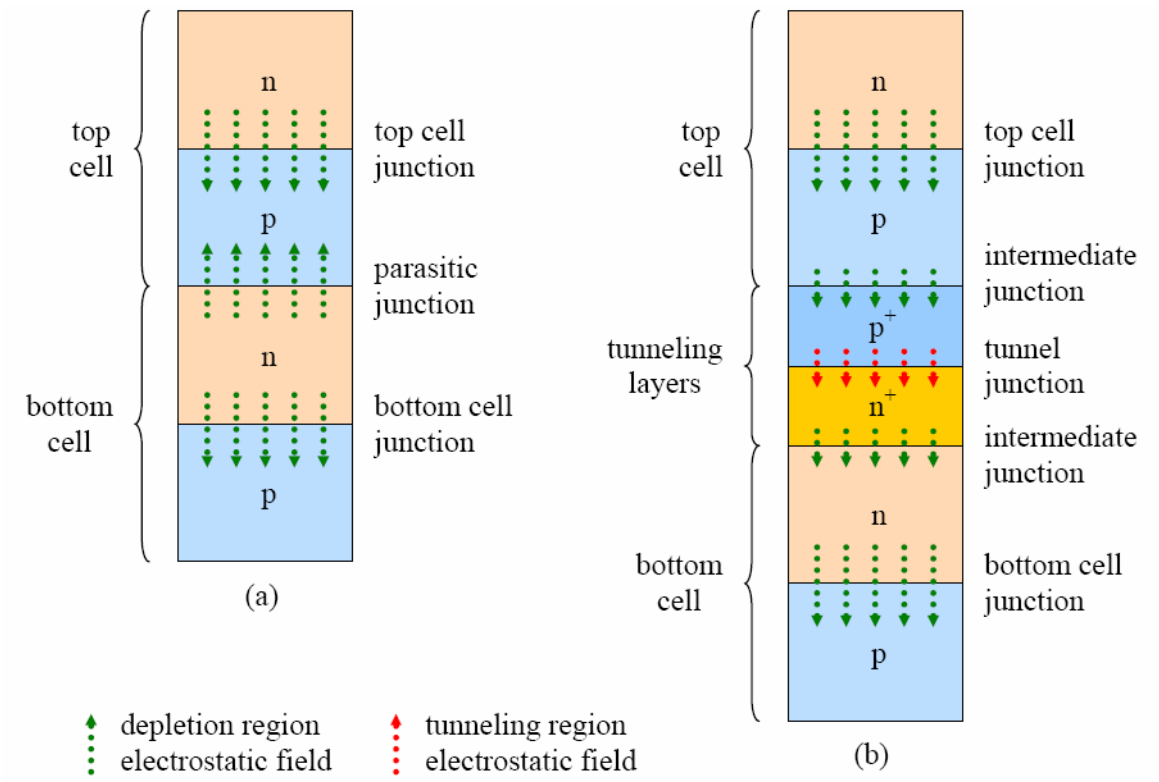


Figure 12. Simple stacking with parasitic junction(a) and tunnel junction (b) [From Ref. 6]

With this tunnel junction in place, there is effectively a series connection between junction-layers which allows current to flow with only a minimal loss in voltage.

b. Materials Incompatibility

Another problem encountered with multi-junction cells is in the compatibility of materials within the manufacturing process. While there have been attempts to mechanically stack cells manufactured separately, the end results were less than optimal. The final cell structure was much thicker and heavier than desired. In addition, losses were encountered by the reflection of light when passing between cells. Consequently, current efforts are on building the entire cell as one unit known as a monolithic multi-junction cell. However, when growing crystals of various materials on top of one another, the materials must have compatible crystal lattice patterns to properly form. In some

instances, a window layer may be grown on the top of the cell in order to bridge some of the material differences, but this does not work in all cases. Consequently, the process of selecting layers for a multi-junction cell must be based on performance criteria as well as materials compatibility.

c. Shadowing Effect

Since layers of a multi-junction cell are stacked on top of one another, light entering a bottom layer of the cell has already been filtered by the layers above it. If the thickness of a top layer is increased, the top layer will have increased performance but the layers below it will consequently receive less light and have reduced photogeneration. Conversely, if the thickness of the top layer is reduced, the top layer will produce less energy but will allow more light to pass through to lower layers.

d. Electrical Limitations

When producing electricity, a monolithic solar cell looks a lot like four dissimilar batteries connected in series. Each battery will have different voltage and current ratings for a given electrical load. When connecting them in series, their overall power production is governed by the following equation.

$$P_{total} = I_{load} \sum V_{junctions} \tag{6}$$

As the number of junction layers increases, the load current becomes a more difficult design concern. In order to harness the maximum power from a junction layer, load current must be close to the junction layer's maximum power point current. However, junction layers based on different materials can have quite different current-voltage curve characteristics, as the examples in Figure 13 illustrate.

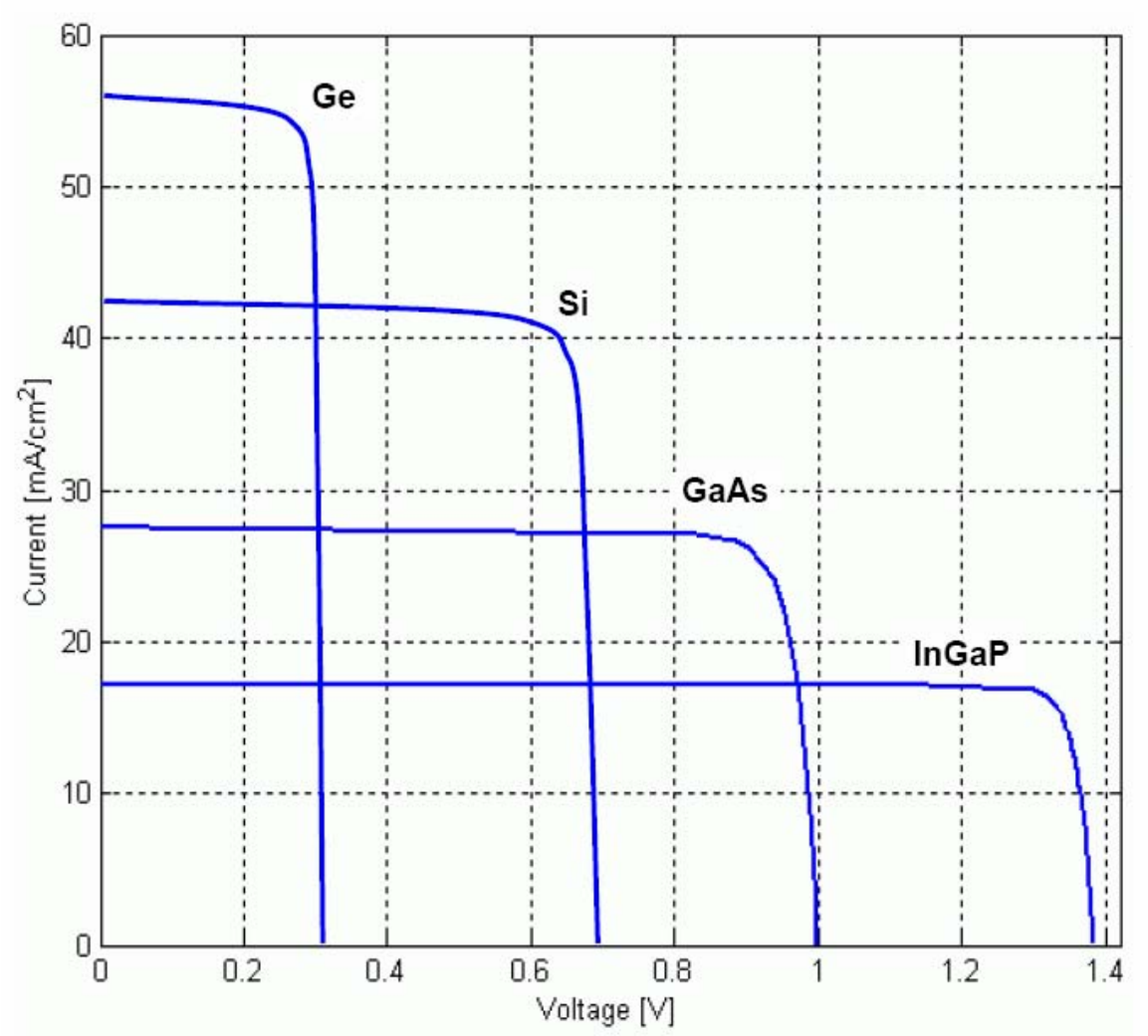


Figure 13. Typical Current-Voltage curves for solar cells based on various materials [From Ref. 6]

In order to obtain the most efficient power output from a cell, all the current values at maximum power need to be matched. The solution to this is related to the shadowing section explained previously. When a cell receives less light, it normally produces an equivalent voltage but lower current. Conversely, when light intensity increases, the cell normally produces more current at an equivalent voltage. As described in the section on shadowing, the thickness of each layer may be adjusted to produce more or less power and allow more or less light to pass through to other layers. By optimally adjusting the thicknesses

of each junction layer, the maximum-power current for all junction layers may be matched in order to get the maximum power output from a multi-junction cell.

C. GENETIC ALGORITHMS

1. Concept and Typical Applications

There are some optimization problems in science to which there only exist complex and computation-intensive solutions. The optimal placement of electrical components on a circuit board or on a chip, the optimal routing of garbage trucks in a large city, and the discovery of optimal robot limb trajectories are just a few examples. In some of these problems, there may exist a method to solve for a solution but it would require too much computation time to be useful in the application. In other types, no one has found a way to directly solve for an optimum solution without first testing every possibility and choosing the one with the best result. Decomposing a problem into its parts and then combining the separate answers will sometimes have unpredictable results. A classic example used to illustrate this situation is the traveling salesman problem. Consider a traveling salesman with a sales area encompassing sixteen cities. In order to make his rounds, the salesman wants to visit all sixteen cities in a single trip. However, since gas prices keep rising, he wants to pick the order to visit cities so that he will travel the shortest path possible. For example, possible routes are 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16, 16-15-14-13-12-11-10-9-8-7-6-5-4-3-2-1, 1-3-5-7-9-11-13-15-2-4-6-8-10-12-14-16, etc. Upon analysis, the total number of routes is N factorial, where N is the number of cities[6]. In our example of a 16 city traveling salesman problem, this is a hefty number.

$$routes = 16! = 20{,}922{,}789{,}888{,}000 \tag{7}$$

However, this number may be cut in half since every sequence of cities has an exact opposite route with the same length. While this would be nice for variety for the salesman, the two routes would have the exact same length. Therefore, the revised number of routes is only ten trillion.

Genetic algorithms represent a class of approximation techniques based on modeling the processes through which organisms breed in nature. An

organism contains genes composed of individual chromosomes which define all aspects of the organism: hair color, skin tone, number of toes, brain size, etc. When the organism breeds with a member of the opposite sex (typically), the genes of the offspring are a mixture of the genes of the two parents. To see how this improves genes over time, we must consider an entire population of organisms. Based on genes, a specific organism has a better or worse chance of both surviving and breeding to produce an offspring. The concept of natural selection is that those organisms with poor combinations of genes are less likely to reproduce. By extension, the population, over time, becomes a mix of organisms which contain only the best genes as handed down from successful parents. Those organisms which contain the bad combinations are more likely to die off without reproducing. In addition, mutations occur spontaneously in nature. Some genetic material is randomly changed by various events. When this improves an organism, the mutation is likely to remain and spread through the population over time. In order to see how this process could be applied to a problem in science, an example of applying a genetic algorithm to solar cell optimization will be given.

2. Illustrative Example: Modeling the Single Junction Solar Cell

Drew Bates, a previous researcher at the Naval Postgraduate School, applied a genetic algorithm to optimize the performance of single-junction solar cells. The following text will explain how the problem in question was modeled using the genetic algorithm. A more detailed treatment of his process may be found in Ref. 7. The first step of applying a genetic algorithm is to define the problem and find a way to represent the problem numerically.

a. The Problem to be Solved

An individual junction layer solar cell actually consists of several layers of material. Within those layers, each region is made of a certain material with a specified thickness and doping concentration. The genetic algorithm used to optimize the cell design was focused on picking the best thickness and doping levels for four of the layers within the cell, namely the window, emitter, base, and back surface field. A sample cell layout showing these traits may be seen in

Figure 14. Note that the bottom and top contacts as well as the cap and anti-reflective coating are not being optimized.

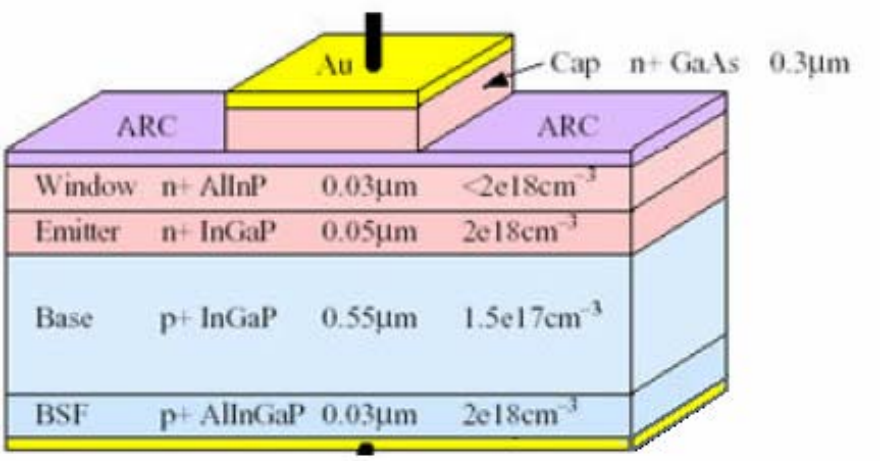


Figure 14. Example of Single Junction InGaP cell [After Ref. 7]

b. Numerically Representing the Problem

One of the methods of implementing a genetic algorithm is to represent trait values using binary strings. In practice, this allows the bits of an individual chromosome to be treated abstractly. Bates designed a 32-bit binary string with which to encode the eight traits with a resolution of four bits per trait.

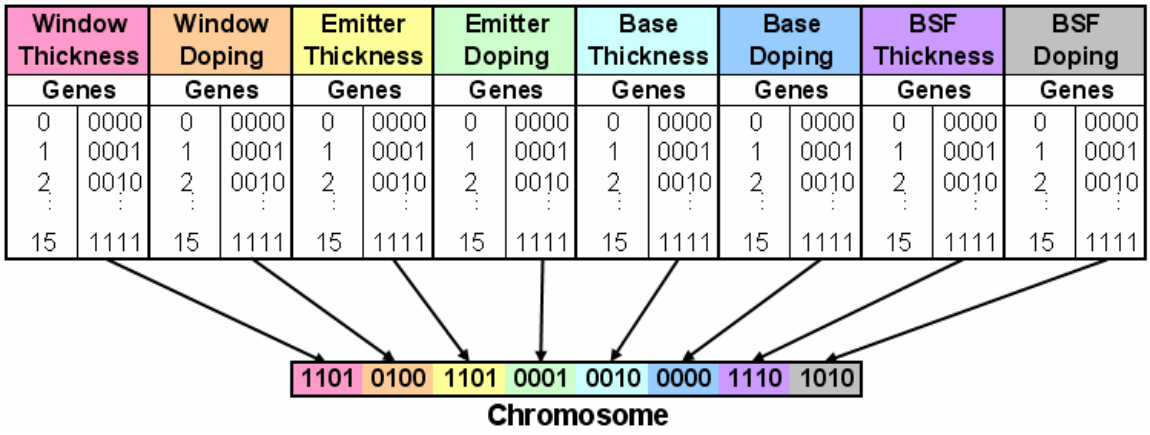


Figure 15. Encoding of traits into a 32-bit chromosome [From Ref. 7]

In practice, however, the base thickness was made a dependent variable and the chromosome was given a dummy bit value for base thickness as a placeholder. This allowed the researcher to specify an overall junction layer

thickness. The base thickness was then calculated by subtracting the window, emitter, and back surface field thicknesses from the specified overall thickness. Quantization, in this case, is the process of assigning trait values to the discrete binary representations. For each trait, a four-bit binary identifier gave the ability to specify 16 different levels for each trait. The following scheme was used for quantization.

	Thickness Constrained		Thickness Variable	
	Minimum	**Maximum**	**Minimum**	**Maximum**
Window Thickness (μm)	0.01	15% of overall thickness	0.01	0.1
Window Doping (cm^{-3})	1e15	1e20	1e15	1e20
Emitter Thickness (μm)	0.01	30% of overall thickness	0.01	2
Emitter Doping (cm^{-3})	1e15	1e20	1e15	1e20
Base Thickness (μm)	N/A	N/A	1	16
Base Doping (cm^{-3})	1e15	1e20	1e15	1e20
BSF Thickness (μm)	0.01	15% of overall thickness	0.01	0.2
BSF Doping (cm^{-3})	1e15	1e20	1e15	1e20

Table 1. Quantization scheme for chromosome encoding [From Ref. 7]

Note: When conducting optimization, all elements other than the above traits of an input deck were held constant. It has been noted that some changes may be needed in input decks for modeling carrier mobility at doping levels as high as 1e20. This should be addressed in further research.

c. Executing the Algorithm

In general a genetic algorithm follows the repetitive process outlined in Figure 16.

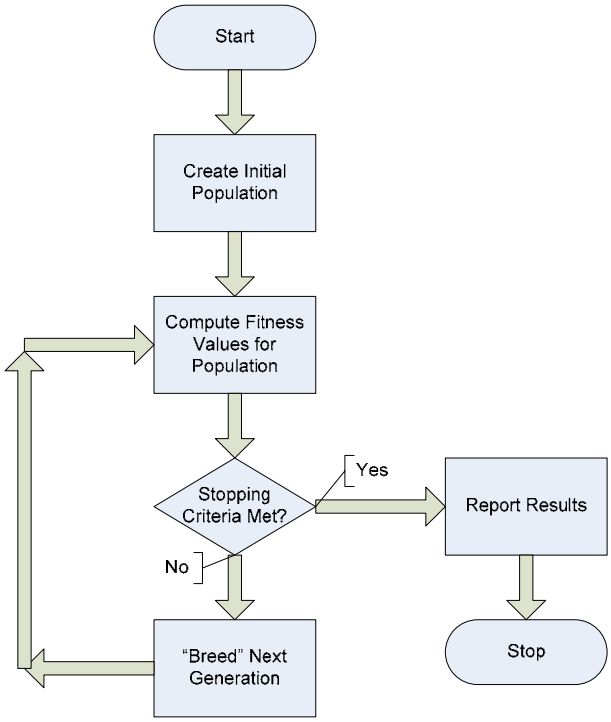


Figure 16. Simplified flowchart for genetic algorithm

To create the initial population, 35 random binary 32-bit strings were generated. In order to compute the fitness value of each chromosome, the trait values were first determined by converting the binary string into real values as shown in the quantization table. These values were then written into a Silvaco ATLAS™ input deck (described in detail in Appendix A). The Silvaco tools provided data on the cell's expected IV curve and therefore the solar cell's expected performance. The maximum output power was used as the cell's fitness value. Once fitness values were determined for all 35 chromosomes, the algorithm evaluated whether or not its stopping criteria had been met. In this case, the genetic algorithm first checked if it had been running for at least 16 generations. If so, it checked to see if the maximum fitness value had changed within the past three generations. If the algorithm had completed 18 generations, the standard was reduced to a 99.9% match in maximum output power for the past three generations. If the algorithm completed 20 generations, it was stopped

regardless of fitness value trends. If none of these stopping criteria had been met, the algorithm then proceeded to breed the chromosomes for the next generation.

d. Breeding a New Generation

In this implementation, breeding took on the process depicted in Figure 17. As will be discussed later, there are many variations in the details of this process.

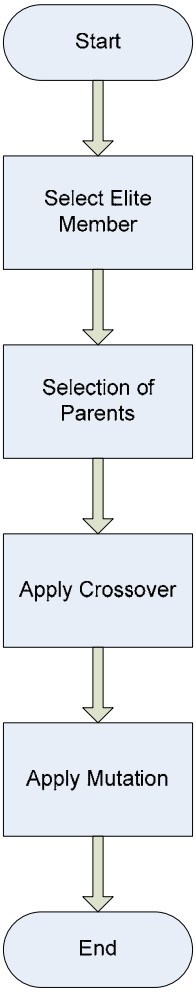


Figure 17. Breeding Process Flowchart

The breeding proceeded according to the following process. First, using an elitist strategy, the best performing chromosome was carried over to the

next generation. Next, a roulette wheel style selection mechanism was employed. Implementation of the roulette wheel was based on two ordered lists. The first list was simply a sorted list of fitness values from highest to lowest. The second list consisted of a sum of a given fitness value and those below it on the list. A random number was then generated between zero and the highest number in the second list. The first value in the second list equal to or higher than the random number indicated a parent to be used in breeding. This process was carried out 34 times in order to generate 17 pairs of parents.

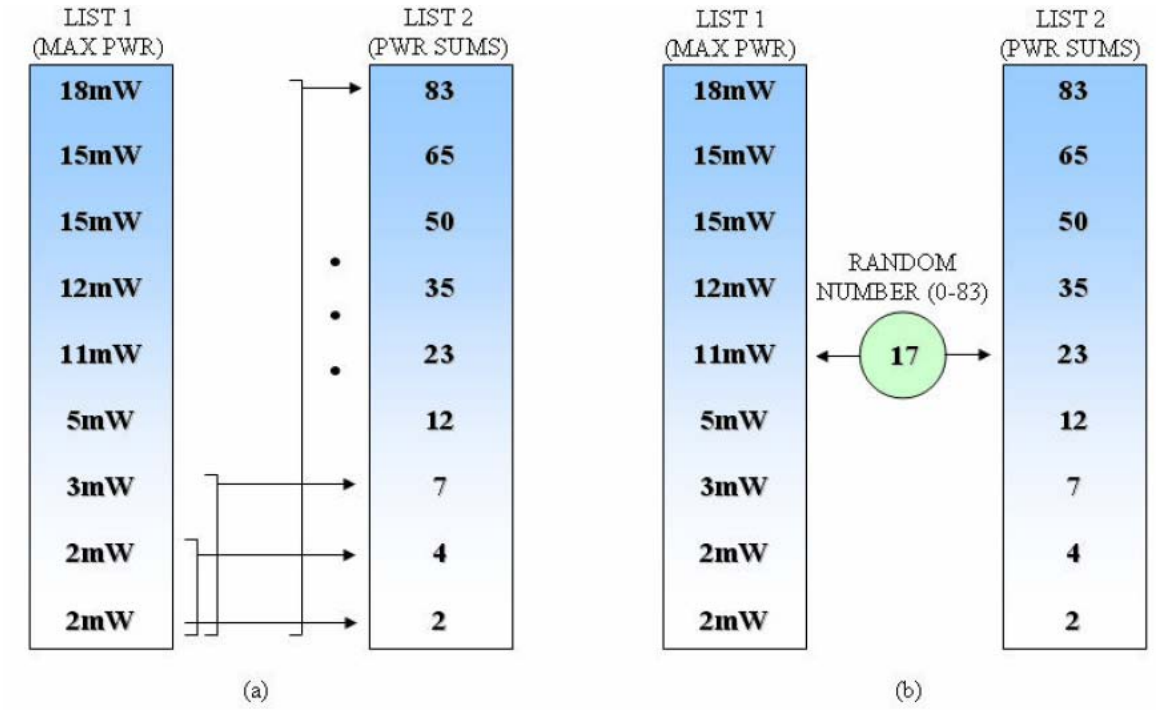


Figure 18. Roulette wheel list generation (a) and selection mechanism (b) [From Ref. 7]

Once parents were chosen, the actual breeding of child chromosomes was handled through a dual-point crossover routine. To imagine the crossover routine, consider the two parent chromosomes laying side by side. In dual-point crossover, two points are chosen along the length of the chromosome. Between those two points, the genetic material of the parents is switched when creating the child chromosome as depicted in Figure 19.

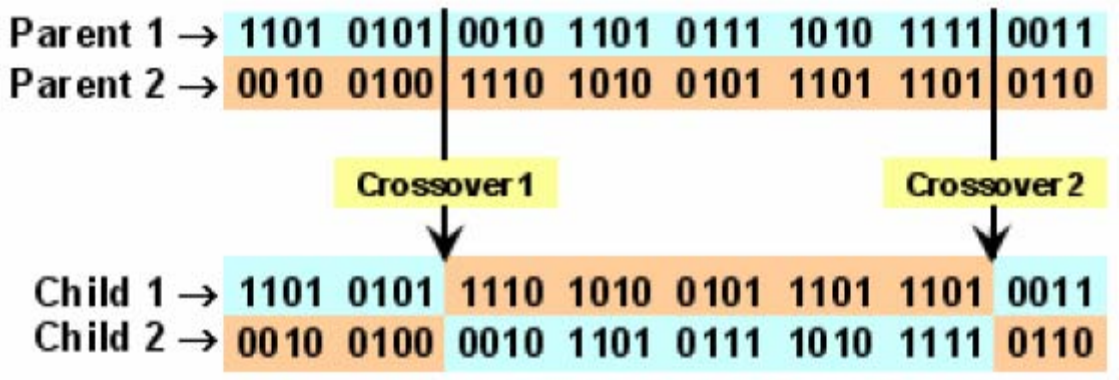


Figure 19. Dual-Point Crossover [From Ref. 7]

The dual point crossover was implemented with 90% probability. This means that in 10% of the operations, the children were simply left as the parent chromosomes. These methods produced the remaining 34 chromosomes for the next generation.

Finally, a one percent probability of a single bit-flip mutation was introduced to allow the algorithm to continually search new areas of the solution space.

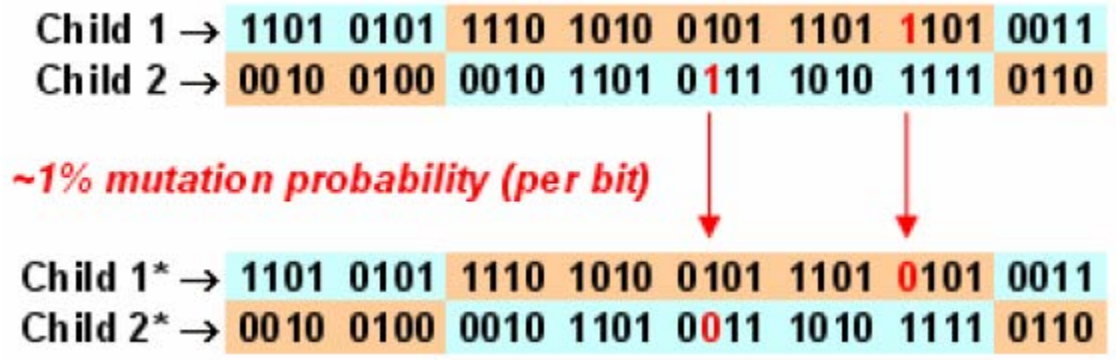


Figure 20. Mutation of Bit Values in a Chromosome [From Ref. 7]

All genetic algorithm procedures, input deck generation, and result analyses were conducted using MATLAB™-based output power analysis tools developed by previous researchers. Results of this process were favorable and

showed, on average, a seven to eight percent improvement in maximum power compared to previous research [Ref. 7].

3. Nuances of Genetic Algorithms

As mentioned previously, genetic algorithms can be implemented in many ways. Holistically, the design of a genetic algorithm is a balance between focusing the algorithm on the solution space and giving it enough randomness to continually search new spaces. If too much structure is given, the algorithm is more likely to get trapped within a local maximum of the solution space. If too much randomness is used, good genetic material such as an ideal trait value may take longer to take hold in a population or may be entirely wiped out through mutation. In this section, a few of the genetic algorithm design considerations relevant to this thesis will be discussed. A very thorough coverage of genetic algorithm design approaches and applications may be found in Ref. 8.

a. Population Size

The small population size used for this implementation was primarily a product of the limited computation power available. However, the population did allow the researcher to ensure that every trait value was represented in at least one chromosome of the initial population [Ref. 7]. In addition, the population size of 30 was specifically mentioned in the text of Ref. 8 along with recommended crossover and mutation settings.

b. Selection Strategy

Roulette wheel selection is one of the most common methods used in genetic algorithm implementations. Ref. 8 categorizes selection strategies according to bias, spread, and efficiency. Bias refers to the probability of selection of a specific individual chromosome. Spread refers to how many times an individual may be simulated. A large spread means the majority of the solution space is equally likely to be selected. A small spread means the algorithm is more likely to in-breed. The efficiency of a particular selection strategy is defined by how efficiently it can be implemented. While selection does take computer time, this time is orders of magnitude less than the amount of time required for simulations. Therefore, efficiency of the selection strategy is

inconsequential in this work. Roulette wheel is classified as zero bias and potentially unlimited spread. Stochastic uniform sampling is another method with zero bias but low spread. Another method chooses parents based on their absolute ranking regardless of the actual fitness values. This is known as a ranking scheme and helps prevent premature convergence of a population [Ref. 8].

c. Crossover Settings

As mentioned, this approach uses a dual-point crossover strategy. This type of a crossover strategy can be implemented with a single point crossover up to one less crossover point than the population size. The latter would end up switching every other bit during crossover. Several papers have debated the merit of different approaches, but dual point seems to be effective in most applications. Another method, known as uniform crossover, creates a randomly generated mask which is the same length as a chromosome. A zero at any point of the mask denotes no crossover while any one signifies that a bit crossover occurs.

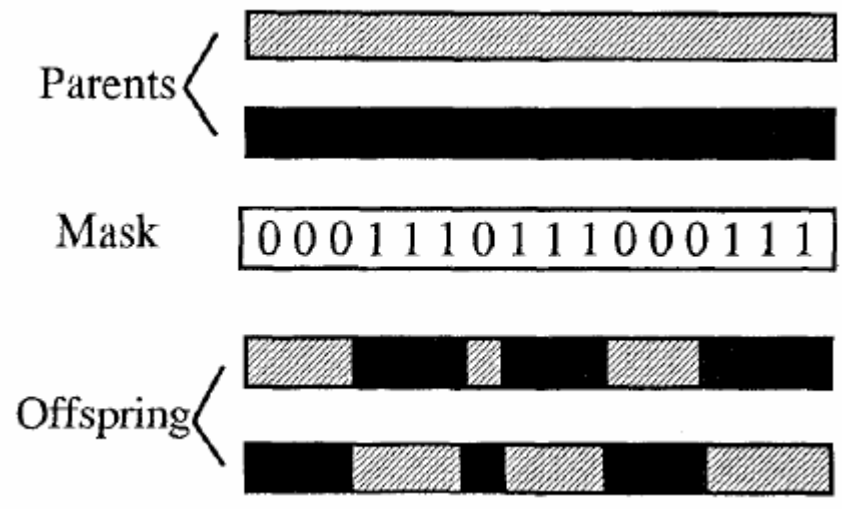


Figure 21. Example of uniform crossover [From Ref. 8]

The selection of a crossover scheme goes back to the holistic view of how much randomness versus how much structure is best for a particular application. In the case of crossover, most schemes only allow crossover to occur at the boundaries between traits. In this way, the basic building blocks of the chromosome are preserved. Genetic algorithm theory predicts that the development and perseverance over time of good genetic material building blocks is often credited with a large part of why the algorithm works. However, in some applications, a greater degree of randomness is desirable to prevent the premature convergence of a population. Uniform crossover, as described, randomly generates a crossover mask which may or may not violate trait boundaries. While there are other crossover schemes, a few have been presented along with the overall tradeoff being managed. For every research paper which showed conclusive results that one crossover method worked better, there is another which states the exact opposite. The consensus is that there probably is an optimum approach to a specific application. However, what works in solar cell optimization might not be the best method for routing garbage trucks in New York City or the placement of electrical components on a circuit board.

d. Probability of Mutation

Similar to crossover methods, the schemes for choosing crossover and mutation probability rates is highly controversial. For mutation, the tradeoff is more simple to understand. At 0% mutation, there will never be any random perturbations introduced into a population. Convergence of the solution set will likely be a one way process and it will be unlikely to escape a local maximum. At 100% mutation, the genetic algorithm simply becomes a random search of the solution space. Therefore, in any application, it is important to observe initial results in order to determine if premature convergence is a problem and then adjust the mutation rate. In many efforts, a linear adjustment is made as the algorithm progresses. In this approach, the mutation rate is slowly increased in subsequent generations while the crossover rate is decreased. Building on these, there are numerous schemes of how to adjust the rates as the algorithm

progresses. Once again, this points back to the original holistic view of trading between randomness and structure within a genetic algorithm implementation.

D. MODELLING CELLS IN ATLAS™ SILVACO

1. Origins of the Cell Model

Until recently, the solar cell industry's only available method of experimentation was to fabricate and then test solar cells. The physical processes involved in cell fabrication make this approach cost prohibitive when many permutations on the design must be tried. However, in the past five years, researchers at the Naval Postgraduate School have been able to accurately model single and multi-junction solar cells within the Silvaco TCAD tool suite[1]. The Silvaco tools create a physical-based model of a semiconductor device within a two or three dimensional space. The physical model includes sizes, thicknesses, doping levels, material properties, etc., of a device. Next, a mesh structure is created within the device to define where analysis is to be conducted. Finally, operating parameters are established and measurements are made by solving a set of differential equations at each mesh intersection. The user can define a device using a standard ASCII text file in a format called an input deck. The TCAD tool suite's DeckBuild™ application may be used to edit, debug, and run input decks. It can also run in a non-graphical batch mode to simulate numerous input decks or as remote sessions. DeckBuild™ parses the input file, builds the physical model of the device, sets parameters, and then calls on the TCAD suite's individual tools as needed to conduct the simulations. In order to best illustrate the flexibility allowed within the TCAD tool suite, Appendix A contains an entire input deck and explains the various settings. In addition, Ref. 9 is the software's user manual.

---

[1] The Silvaco TCAD tool suite allows physically based models of semiconductor devices. It is used in modeling a wide range of electronic devices. To learn more, visit Silvaco's web site at http://silvaco.com/products/TCAD.html last accessed September 2006

2. Previous NPS Research Progress

a. Michalopoulos

Panayiotis Michalopoulos was the first researcher at NPS to identify the Silvaco TCAD suite as a potential method of modeling solar cells. He developed the first single junction cell models and validated them against published results. Further, he continued by modeling well-documented dual junction cells and validating them against published results. When constructing the cells, Michalopoulos was able to model and test a tunnel junction between layers of the cell. Finally, he modeled and optimized a triple junction solar cell and validated the results against published performance data. However, in the triple junction cell, the publication used did not give the actual physical structure used in cell construction. Michalopoulos was able to predict cell construction based on lessons learned in previous simulations and his performance data closely matched the published results. In building the original model, Michalopoulos conducted extensive analysis of publications to best define the material properties of the relatively exotic materials used in single and multi-junction cells. The results of his research are published in his master's thesis [Ref. 10].

b. Green

Max Green, another NPS researcher, conducted an extensive validation process on Michalopoulos' work while re-creating the Silvaco cell models. Although the majority of the cell configurations were validated, the tunnel junction model was found to be incorrect. Michalopoulos' tunnel-junction model was not set up correctly and did not have the correct current-voltage characteristic curve. When the construction was corrected, the model did not function correctly. No researchers have since been able to get the tunnel junction working correctly. A thorough discussion of this challenge is given in Chapter VII Part D of his thesis. Green continued his validation work by mechanically stacking the cells. His final step was to construct a four-junction

cell based on adding an InGaNAs layer. The cell was modeled and it's theoretical output levels were computed. The results of his research are published in his master's thesis [11].

c. Bates

Drew Bates, yet another NPS researcher, pursued two major research thrusts: (i) cell optimization and (ii) using different light spectra. He started by designing a genetic algorithm for use in optimizing each junction layer. After realizing improved performance in each layer at various thicknesses, Bates developed an iterative current-matching technique for adjusting the thickness of each junction layer in order to maximize overall cell output. The iterative current matching technique improved simulated cell performance. Bates final work was to optimize the design of a triple-junction cell under the Martian light spectrum. As predicted, a cell optimized for Earth orbit is not optimally tuned for performance on the Martian surface. By adjusting thickness and doping levels from an Earth-optimized cell, better performance can be obtained under Martian conditions. The results of his research are published in his master's thesis [Ref. 7].

d. Crespin

A fourth researcher at NPS to work in this research area was Aaron Crespin. One of the primary drivers in spacecraft solar array design is the loss in array efficiency caused by radiation effects. Crespin successfully modeled radiation effects in a single-junction Gallium-Arsenide cell using Silvaco ATLAS^{TM}. The cell's degraded performance with the radiation effects closely matched published research showing results from experimenting with real cells. Extension of Crespin's work could potentially lead to optimized cell designs which degrade gracefully over a spacecraft's life despite the effects of the cumulative dose of radiation exposure. The results of his research are published in his master's thesis [Ref. 12].

THIS PAGE INTENTIONALLY LEFT BLANK

III. PREVIOUS OPTIMIZATION APPROACHES AND THE CASE FOR DISTRIBUTED COMPUTING

A. DREW BATES' GENETIC ALGORITHM AND ITERATIVE CURRENT MATCHING APPROACH

Drew Bates approach to optimizing single-junction solar cells is the use of a genetic algorithm as outlined in Chapter II. His results found optimal configurations of the individual junction layers for several cell types and thicknesses.

The method used for assessing the quad-junction cell was to subject the cell to an iterative current-matching. The routine begins with a multi-junction cell with thickness values slightly larger than the expected optimum values. The routine then evaluates the current-voltage curves of each junction layer by comparing the short-circuit currents. Short-circuit current was initially used as an approximation of a junction layer's maximum power current in order to save computation time. Junction layers were initially paired up with the top two and bottom two layers together. In order to match current within the pairs, parametric analysis of thicknesses for the upper and lower half of the pair were made. Once the upper and lower pairs' current converged within 99.6%, the second and third layer currents were compared for one iteration, followed by adjusting the thickness adjusted to match currents for the two pairs. This process was repeated until all four junction layers were within 99.6%. At this point, the routine switches mode, from matching short-circuit current to matching max power current. The routine ends once all four max power currents are matched. Pseudocode for this routine is listed below. Note that in this pseudocode the symbol <> denotes the lack of convergence within 99.6%.

procedure iterative_current_match
    var
        I_{sc1}, I_{sc2}, I_{sc3}, I_{sc4}, I_{mp1},I_{mp2},I_{mp3},I_{mp4} : **double**
        thickness1,thickness2,thickness3,thickness4 : **double**
    begin
        Initialize *thickness1,thickness2,thickness3,thickness4* to values well

above their expected optimal thicknesses

while $I_{sc1}<>I_{sc2}<>I_{sc3}<>I_{sc4}$
{in other words, while the currents have not converged}
I_{sc1}, I_{sc2}, I_{sc3}, I_{sc4}=simulate cell(*thickness1,thickness2, thickness3,thickness4)*
while $I_{sc1}<>I_{sc2}$ or $I_{sc3}<>I_{sc4}$
{This part of the loop adjust thicknesses between pairings of the upper two and lower two layers}
if $I_{sc1}>I_{sc2}$
Reduce *thickness1*
end;
if $I_{sc1}<I_{sc2}$
Increase *thickness1*
end;
if $I_{sc3}>I_{sc4}$
Reduce *thickness3*
end;
if $I_{sc3}<I_{sc4}$
Increase *thickness3*
end;
I_{sc1}, I_{sc2}, I_{sc3}, I_{sc4}=simulate cell(*thickness1,thickness2, thickness3,thickness4)*
end;
{Once pairings have been current-matched, now adjust thickness to match the two pairings}
if $I_{sc2}>I_{sc3}$
Reduce *thickness2*
end;
if $I_{sc2}<I_{sc3}$
Increase *thickness2*
end;
end'
while $I_{mp1}<>I_{mp2}<>I_{mp3}<>I_{mp4}$
Same as above loop but comparing max power current vice short circuit current
end;
end;

The current matching routine successfully increased the quad-junction cell's power output by approximately seven percent. Figure 22 illustrates the convergence of individual junction current levels as the routine progressed over 100 iterations.

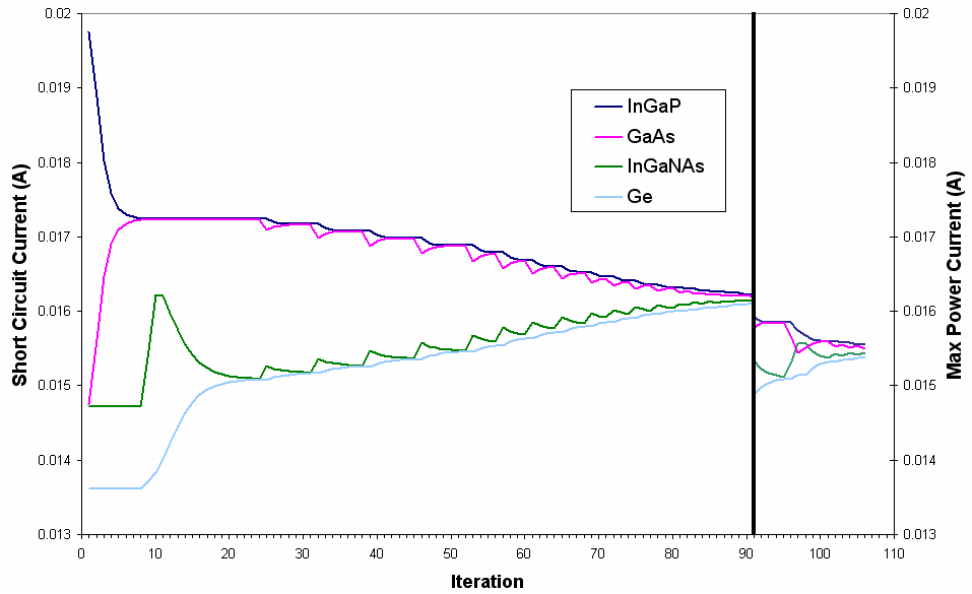


Figure 22. Results of Bates' iterative current matching routine for four-junction cell [From Ref. 7]

B. THE CASE FOR DISTRIBUTED COMPUTING

1. Size of the Solution Space

The genetic algorithm approach used by Drew Bates on each junction layer creates a large space in which to search for a solution. Each cell configuration or chromosome is represented by a 28-bit binary number. Each bit has two possible values, zero or one. The total size of the solution space turns out to be 2^{28} [Ref. 13].

Bates used a single Pentium IV 2.52 GHz computer with 1GB of RAM to conduct simulations. Using this platform, he found that it was possible to run a single simulation every 2 minutes of computer time. Using this method, an exhaustive search of the solution space would take an unacceptably long time, roughly 13,003 years.

In addition, sometimes the ATLAS™ simulations would produce corrupted data causing the system to halt; restarting the simulation required manual intervention. Bates genetic algorithm implementation had a population size of 35 and was halted if it did not converge within 20 generations. Assuming each

member of each generation was unique, this would only cover 700 of the possible 268,435,456 solutions. In addition, the genetic algorithm re-introduces chromosomes by breeding and mutation. Bates observed that each optimization run only tested approximately 300 unique chromosomes and would take approximately 24 hours of computation time without runtime errors.

However, testing such a small portion of the solution space begs the question of whether or not the algorithm converged in a local maximum. Most research papers on genetic algorithms will show results up to 500 generations depending on the complexity of the problem. In order to better search the solution space, more computation power was needed.

Two possible approaches to speeding up this process are to purchase or build a faster machine to do the work or to distribute the computational tasks among many computers. The latter approach was chosen as described below.

2. Distributed Computing Approach

By Flynn's taxonomy, computing algorithms may be classified as Single Instruction Single Data (SISD), Multiple Instruction Single Data, Single Instruction Multiple Data, and Multiple Instruction Multiple Data [Ref. 14]. A commonly used derivative of this is Single Program Multiple Data (SPMD). When comparing algorithms using these distinctions, an engineer gains insight into the best way to implement a distributed computing application. As an example, a SISD is represented by a desktop personal computer running applications in series. In the case of simulations of solar cells, SPMD is an appropriate model to apply since the majority of computation time used by Silvaco ATLAS^{TM} is for simulating the solar cell design. At this time, the software is not designed to split a single simulation over multiple processors. The multiple data aspect describes the numerous cell designs to be simulated. Accordingly, a simple and ideally distributed computing algorithm is to centrally manage the assignment of simulations to a large number of separate computers, which could in turn run simulations locally and report back their results.

3. Choosing a Distributed Computing Platform

Next, a search was initiated to find what types of distributed computing software were available to support this approach. While numerous approaches were found on the web, many of them required the use of a programming language unique to the platform. Consequently, these were not considered and the focus of the search was placed on systems which allowed the use of fairly standard programming languages with minor modifications to allow the coordination of numerous machines. Four primary systems were considered and each is briefly described with a comparison chart following the descriptions.

The Berkeley Open Infrastructure for Network Computing (BOINC) is an open source system designed for distributing computing work across the internet [Ref. 15]. It was based on the Search for Extra Terrestrial Intelligence (SETI) at home project which distributes radio telescope recordings to volunteer computers worldwide which then conduct signal analysis on the recorded data and report back their results [Ref. 16]. By downloading a simple client, a user may define the amount of computer resources which may be used (idle time only, memory, hard drive, cpu percentage, etc) and enter the web link for projects they wish to participate in. The client then logs into the projects, downloads any needed software and data files, and then commences work according to the user preferences. Users may also participate in several projects and define the amount of computer resources devoted to each project. The project sponsor uses a different version of the BOINC software which tracks users, data files, software versions, etc. This server software is not simple to install but there is ample help from UC Berkeley and other established projects available. The limiting factor, however, is that the solar cell simulations require use of the proprietary Silvaco TCAD suite of software. Freely distributing this to volunteers across the internet would be illegal.

The next two systems considered were the Parallel Virtual Machine (PVM) [Ref. 17] and Message Passing Interface (MPI) [Ref. 18] packages available for Unix/Linux and Windows platforms. While they differ in implementation, they

both offer a similar approach to a distributed computing solution. When installed, the PVM or MPI software acts as a buffer between the distributed programs and the operating system on each machine being used. The packages handle the housekeeping functions necessary to coordinate numerous machines. These functions include the passing of data (messages), booting and shutting down processes on a number of computers, process monitoring and control, etc. The programmer may write in C/C++ or Fortran and compile the code using a slightly modified compiler which includes the commands for accessing all the distributed computing functions. PVM's tutorial's were last updated in 1997. The LAM MPI web site was current with a full, proctored, free tutorial which was updated in 2006. Both LAM-MPI and PVM are available as installed features within Redhat Enterprise Linux (RHEL) and Fedora (the free version of RHEL).

MATLAB™ has recently developed a distributed computing toolbox which allows some of the functionality needed. However, at the time this research began, the toolbox had been recently released and was still in heavy development. The other downside is that MATLAB™ is a commercial product and has a significant cost. Depending on home many clients are to be run, this can range from $2,000-$5,000 [Ref. 19]. The preferable approach is to have processes run in the background. Their second release of the toolbox appears to be more fully featured and allows processes to run in the background. The toolbox is based on the MPI packages adapted for use within MATLAB™.

	Maturity	**Learnability**	**Failure Tolerance**	**Programming Language**	**Cost**
BOINC	2	2	3	Any	Free
LAM	5	5	4	C++/Fortran	Free
PVM	5	4	4	C++/Fortran	Free
MATLAB™	2	5	1	MATLAB™	$2,000-$5,000

Table 2. Comparison of Distributed Computing Approaches

While weighing this decision, two additional factors were introduced. First, a new lab was built with 18 dual processor machines with ample memory running on Linux. The lab was built for the Oceanography department and was not yet being utilized. Second, the school was becoming increasingly concerned about energy conservation. This meant that the majority of lab machines were remotely shut down at night. Since the majority of Linux and Unix users work remotely, the Linux and Unix machines are generally exempt from this automatic shutdown. With the pre-installed packages with free compilers and development software available under Linux, the lab was the best candidate. LAM-MPI was chosen over PVM because of the more recent training products and its continued development and support on the Web.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. RESULT VALIDATION

A. DISTRIBUTED COMPUTING IMPLEMENTATION

1. Hardware Setup

The platform used for running the optimization application consisted of a network of 18 dual-Xeon processors, each with 1GB RAM. In addition, two Pentium III desktop computers handled administrative functions: One was used as a file server for all data and working directories, while the other was used as a distributed-system monitor [Ref. 20].

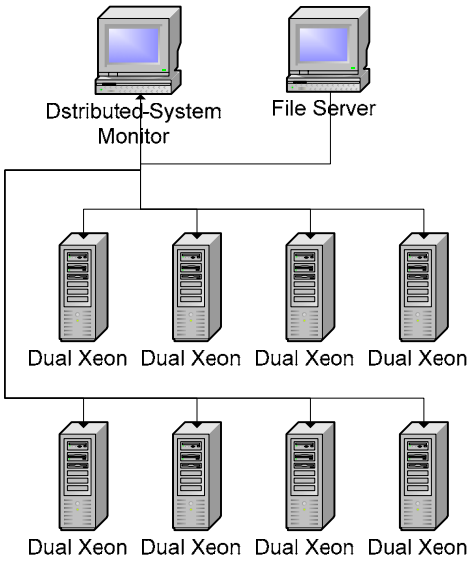


Figure 23. Original Lab Setup

2. Software Setup

a. Operating Systems

All lab machines were configured with Red Had Enterprise Edition Linux (RHEL) and received occasional upgrades from IT support personnel. The two desktops used for file sharing and software development were set up with

Fedora Core 3 (FC3), an open source operating system which functions as a testbed for development for RHEL. Typically Fedora Core releases test newer features and work out the bugs before software is incorporated into the more stable RHEL operating system. Since the Fedora Core series has many more "bleeding edge" packages, it provided a more feature rich and easy to support system for software development. The core packages were similar enough to interoperate without problems. However, the LAM MPI packages for FC3 had developed up to version 7.1.1-7 while RHEL only had version 6.5.9-1. The two versions were not compatible. When software was compiled on the development desktop, it would not execute on the computing platform. An older package was located for the development desktop and reverted to version 6.5.9-3. While a slightly different version, they proved compatible in execution.

b. Distributed Computing Scheme

The distributed-system monitor and hill-climbing optimization application were written in the C programming language. C's primary advantages were familiarity to the author, numerous online tutorials and references, and the ease of low-level process control. The software was composed of a few thousand source lines of code. Some highlights of the code follow. A more detailed treatment is given in Appendix C.

The master node of the distributed computing system was designed to take a few inputs. The first was a file giving general simulation run information. This included the location of input files, type of cell to be simulated, location to store result files, how many times to retry failed simulations, how much time to allow a simulation before aborting it, and other housekeeping parameters. The second input was a set of three files which define the chromosomes to be simulated, those which have already been successfully simulated, and those which encountered simulation errors. Bates' research identified unique chromosomes by their decimal number. These three files were simply ASCII encoded lists of chromosomes, one per line. This system allowed a simulation to be resumed in case of system software error. The master node's

job was to read the input files, build the ATLAS™ input decks, assign chromosomes to clients for simulation, track overall progress, and consolidate the results upon completion.

The client node of the distributed computing system had two primary inputs. The first input was housekeeping data similar to the master node. The second input was the client's assignment from the master node. The client took advantage of the C language's facilities for low level process control. The client receives an assignment and then spawns a new child process to enact the simulation by running ATLAS™ on the assigned file. The original parent process periodically checks the operating system's record of the simulation process status to ensure it is still executing correctly. If the simulation went past a specified time threshold, the parent process would kill the simulation and send a failure notice back to the master node. If the simulation completed successfully, the client would extract the data from the ATLAS™ output file, check it for validity, and save it in a more compact form for further analysis.

B. SINGLE-JUNCTION RESULT VALIDATION

1. Coarse Sampling and Gradient Ascent Method and Results

The first concern with Bates' data was that it became trapped in a local maximum and failed to adequately search the solution space. Through the use of distributed computing, a coarse sampling and gradient ascent method was first used. The strategy implemented in the hill climbing method was to first conduct a coarse sampling of the solution space and then execute a gradient ascent method on the best candidates found in this process. The following pseudocode outlines this process.

```
procedure coarse_sample_and_gradient
    var
        list_of_chromosomes      :   array of 2187 int
        results,gradient_resutls :   array of 2187 int
        gradient_candidates      :   array of 5 int
        candidate,new_candidate  :   int
        loopcounter              :   int
    begin
        list_of_chromosomes=permutations(all trait values 1,8,15)
```

```
        results=conduct_simulations(list_of_chromosomes)
        gradient_candidates=pick_best_five(results)
        for loopcounter=1 to 5
            candidate=gradient_candidate(loopcounter)
            while candidate≠new_candidate
                list_of_chromosomes=permutations(all trait values of
                    candidate as well as plus and minus 1)
                gradient_results=conduct_simulations(list
                    _of_chromosomes)
                new_candidate=maximum_of(gradient_results)
                if new_candidate>candidate
                    candidate=new_candidate
                    new_candidate=0;
                end;
            {if no improvement was found, the gradient loop exits, if an
                improvement was found, the gradient process repeats
                for the new candidate}
            end;
            store_newfound_local_optimum
        {algorithm would now loop and conduct the hill climbing on the next
            candidate}
        end;
    end;
```

The first consideration was how coarse of a sampling of the solution space to make. The initial assumption was that an optimum position within the 15 trait positions cannot be predicted. On one end of the spectrum is testing only one value of each trait and the other end of the spectrum is testing all combinations of every value of every trait. The following table shows how increasing the level of granularity results in an exponential growth in computation time. The following table shows the computation time required to achieve varying degrees of granularity in the coarse sampling. The table assumes two minutes per simulation and that the distributed computing system will consistently employ all processors with three simulations running on each simultaneously. However, our ability to run simulations was limited by 50 licenses of the Silvaco software and that not all processors were always available. In addition, some re-tooling was required in order to change between cell types. Within a single cell type, a hill climbing approach was applied to the top 5 results of the coarse sampling and each hill climbing application would normally run for five iterations or more.

Cell Types	Traits per cell	Values per trait	Total Permutations	Single Computer Computation Time				Distributed Computing Computation Time			
				y	d	h	m	y	d	h	m
20	7	1	20	0	0	0	40	0	0	0	0.741
20	7	2	2560	0	3	13	20	0	0	1	34.81
20	7	3	43740	0	60	18	0	0	1	3	0
20	7	4	327680	1	90	2	40	0	8	10	16.3
20	7	5	1562500	5	345	3	20	0	40	4	30.37
20	7	6	5598720	21	111	0	0	0	144	0	0
20	7	7	16470860	62	246	4	40	1	58	15	11.85
20	7	8	41943040	159	219	5	20	2	348	18	45.93
20	7	9	95659380	364	0	6	0	6	270	9	0
20	7	10	200000000	761	12	18	40	14	34	0	47.41
20	7	11	389743420	1483	15	7	20	27	169	6	21.48
20	7	12	716636160	2726	338	0	0	50	182	0	0
20	7	13	1254970340	4775	139	8	40	88	158	1	2.963
20	7	14	2108270080	8022	122	21	20	148	205	1	17.04
20	7	15	3417187500	13002	363	18	0	240	290	15	0

Table 3. Computation Time required for varying granularity of coarse searches of the solution space

For time's sake, the three-values-per-trait approach was used. In order to do this, trait values zero, eight, and 15 were used for each trait. This was chosen to help test if the quantization range applied to the trait values was correct and to give the widest possible solution space. Upon completion, the results shown in Table 4 were obtained. In the table, the optimum found by Bates is listed followed by the data from the best result using the methods outlined above. Since Bates results were obtained under Windows-based Silvaco software, the best results obtained under Linux were converted into Windows format and simulated in the same manner used by Bates for consistency. The power values in the tables were all generated under Windows.

Coarse Sampling and Gradient Data for InGaP					
Configuration		Total Simulated	Power(W/cm^2)		
Cell Type	Thickness		Bates	Hill Climb	% diff
InGaP	0.25	175707	0.018398885	0.018224933	-0.945%
InGaP	0.50	99239	0.023158565	0.021891432	-5.472%
InGaP	0.75	46691	0.025264187	0.024779541	-1.918%

Table 4. Coarse Sampling and Gradient Ascent Data for InGaP Cells

The disparity in total number of chromosomes tested has to do with the software development process. The first configurations tested ran significantly more times than later configurations because the distributed computing software was still under development. A feature implemented early on was checkpointing to aid in recovery from system or application failure. Other cell types tested showed similar results to Table 4. In almost every cell type, the gradient ascent process for the five best candidates led to five separate local maxima. In addition, the local maxima were all of lower output power than the results obtained by Bates using the genetic algorithm. This indicates that the solution space is likely riddled with local maxima and that the traits do not all have a linear effect on cell output power.

2. Distributed GA Method and Results

Upon completion of the hill climbing method, the next logical step was to extend Bates' work by using the genetic algorithm with the aid of the distributed computing system. In order to simplify the GA portion of the programming, the MATLAB™ Genetic Algorithm and Direct Search toolbox was used. The toolbox allows the rapid re-configuration of population, selection, crossover, mutation, and most genetic algorithm properties without manual intervention and re-coding of an application. Originally, the settings specified in Bates' thesis were used. His implementation used a uniform (random) initial population of 35, four-bit per trait representation, roulette wheel style selection, dual-point crossover, and a single chromosome elitist strategy. While the majority of these settings are implemented in the MATLAB™ toolbox, there are some minor differences. For a more detailed treatment, see Appendix C.

For each cell configuration, the MATLAB™ toolbox was set up to run for 50 generations before stopping to report results. In order to take advantage of the distributed computing architecture already built, a MATLAB™ routine was written to store cell configuration data in a file and then send messages (via a single one or zero in a file) to the distributed computing master node. When all simulations were complete, the distributed system monitor would compile the

results and then similarly signal MATLAB™ that the results were ready. MATLAB™ would import the data, breed the next generation, and continue the process. In order to introduce MATLAB™ to the lab, a separate PC was used.

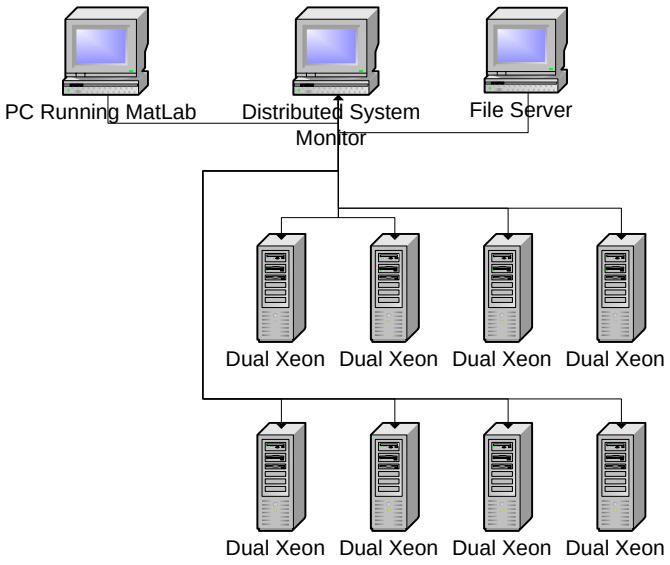


Figure 24. Revised distributed computing platform

Early iterations of this process showed that the population would quickly converge with little mutation by generation 15 and would show little improvement or change from that point through generation 50. In order to better search the solution space, the mutation rate was increased to ten percent. In most cases, this type of tuning resulted in small incremental gains throughout the 50 generation optimization. This 50 generation approach with the increased mutation rate was used on all 14 cell configurations.

Cell Type	Thickness	20 gen	50 gen	% diff
InGaP	0.25	0.018398885	0.018401961	0.017%
InGaP	0.50	0.023158565	0.022275164	-3.815%
InGaP	0.75	0.025264187	0.025273847	0.038%
GaAs	0.50	0.024628965	0.024722282	0.379%
GaAs	1.00	0.028179029	0.028189460	0.037%
GaAs	1.50	0.029342492	0.029344780	0.008%
GaAs	3.00	0.030039575	0.030073571	0.113%
GaAs	5.00	0.029878681	0.029982369	0.347%
InGaNAs	1.03	0.018074821	0.018174204	0.550%
InGaNAs	1.55	0.018531643	0.018581991	0.272%
InGaNAs	2.06	0.018633871	0.018668836	0.188%
InGaNAs	4.00	0.018461029	0.018497504	0.198%
InGaNAs	6.00	0.018428607	0.018575130	0.795%

Table 5. Extended Genetic Algorithm Results through 50 Generations with Increased Mutation Rate

C. MULTI-JUNCTION CELL APPROACHES

1. GA Real-Values Method and Results

Adapting distributed computing to the optimization of the combined quad-junction solar cell is a more complex problem than the GA for single-junction cells. Bates used the genetic algorithm on each of the individual junction layers and then conducted an iterative current-matching routine on the combined cell. Results of the current-matching routine were then used to choose the next thickness of individual junction layers to optimize. The first question which comes to mind is why the genetic algorithm was not applied to the entire cell. While time may have been a limiting factor, the optimization process itself combined with the current multi-junction cell model is a limiting factor. In the modeling of the genetic algorithm on each single-junction, it was assumed that the thickness values of individual layers in a junction would fall within certain overall thickness ranges. Accordingly, the quantization process assigned thickness values based on these percentages to a fixed binary mapping. Applying this type of process to an overall multi-junction cell would be limiting. In addition, the programming of such an approach would be quite tedious.

The MATLAB™ GADS toolbox has the ability to represent chromosomes and traits as bits or as real valued numbers. If binary is used, the user must provide their own functions for decoding and encoding the chromosomes for use in the fitness function. Using the MATLAB™ real number feature allows the numbers to be directly applied. In addition, the user may specify bounds for real valued numbers. This accomplishes the bounding used in Bates' quantization without the artificially constraining bit increments to values; the toolbox also allowed for a much simpler programming approach.

The final experiment of this thesis was to attempt such an approach. Since the distributed computing architecture was designed to take a file listing chromosome numbers as its argument, significant re-tooling was required in order for it to take an input of real valued numbers. This also presented a dilemma of how to store results. With a 28-bit chromosome used in single junction cells, each chromosome could be represented by a unique number which was realizable in most computers using an unsigned integer. When using real values, the solution space is infinite. Consequently, individual chromosomes were identified only by generation and a sequential list of numbers within that generation.

Using a random initial population, the simulations seemed to all get an identical power output value. When the trait data from Bates' previously optimized cell was introduced, it would get one (higher) output value and the remainder of the chromosomes would have an identical (lower) output value. The reason for this is not certain, but there are a few explanations to consider.

2. Possible Explanations for the Results

The first is that the comparisons being made are at a high level of detail. On individual junction layers, comparisons between optimal designs differ in the microwatt range. Multi-junction cells of similar construction differ in the low milliwatt range. It is possible that the randomly generated cell properties produced cells whose output power does not differ that significantly. However, as successive generations mixed attributes of the optimized cell and the truly

random ones, no intermediate power output values were found. This is contrary to expected results. Another explanation is that error is introduced in the way the current-voltage curves are measured and by the way the current cell model mechanically stacks the cells.

Without a tunnel junction, the method of measuring overall cell output power is not straightforward. For each junction, current-voltage solution points are chosen based on percentages of the short-circuit current of that junction. The detail level of these points is significantly higher around the knee of the curve in order to gain a more detailed current-voltage curve for the junction. However, for the lower junctions of the cell, this may not correspond to the operating point when limited by the current of the top junction of the cell. Once the current voltage curves for each junction have been obtained, a MATLAB™ code segment titled mj_ivmaxp.m calculates a max power for the cell. Since the current of all junctions is limited by the junction which produces the least current, the majority of layers will not be operating at their maximum power point at which the current cell models were designed to give the greatest amount of detail. Instead, the routine takes the data available and conducts linear interpolation between points as needed to find a junction layer's performance at the correct operating current. The lack of detail in measurement at this current level combined with the linear interpolation may introduce significant error and loss of detail. When comparing cell performance on the levels of hundredths of a percent, this level of detail is important. Figure 25 shows the four individual junction layer IV curves on a single plot. The IV curves are shown and superimposed with the individual data points on which the curve was fitted. In the case of this cell, the InGaP layer would have the limiting current. It is notable that the other three curves do not have any measured points at that current level.

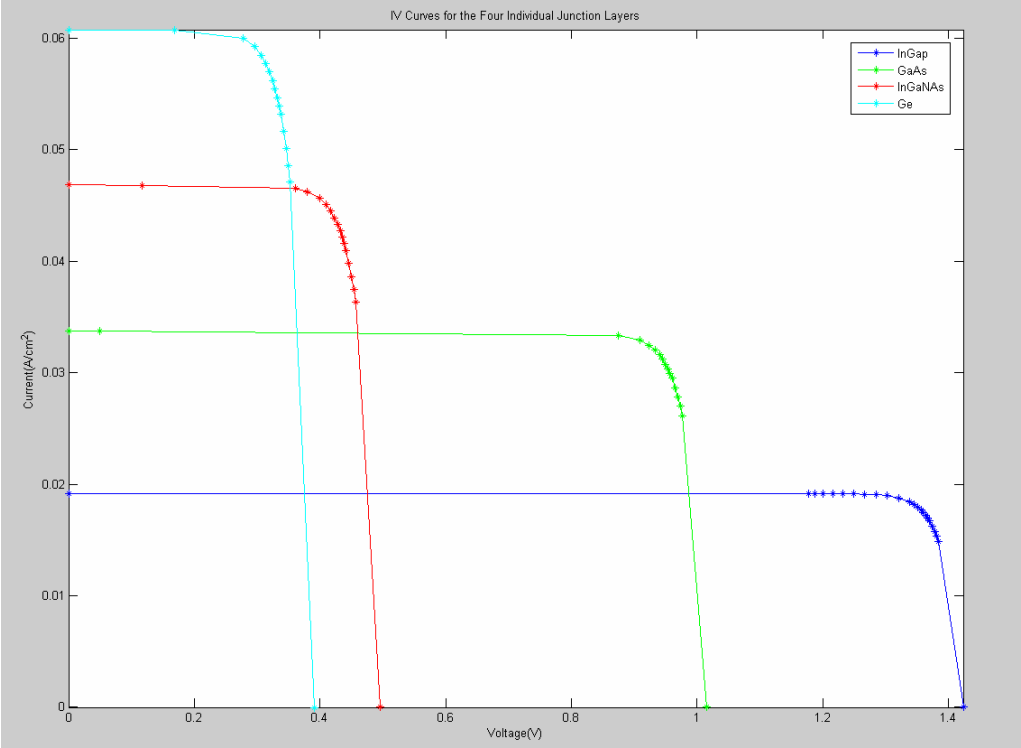


Figure 25. Highlighted Measurement Points for Individual Junction Layers

The accomplishment of a working tunnel junction model in Silvaco's software could greatly benefit this optimization work. First, the tunnel junction would allow the software to simply measure the total cell output when subjected to light without regard to the individual junction layer contributions. This would reduce the complexity of evaluating the output power and allow better integration into a fitness function to enable genetic algorithm optimization. Second, the properties of the tunnel junction itself could be optimized to give the best total output power for the multi-junction cell.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS

In this thesis research, a coarse sampling/gradient ascent algorithm and extended genetic algorithm were used to test the validity of Bates' optimization approach. The results of applying the coarse sampling/gradient ascent algorithm did not yield any improved power output for cells. The extended genetic algorithm runs with 50 generations and increased mutation rates consistently produced improved results. Thus, the results of this experiment provide confirming evidence for the hypothesis that the procedure used by Bates produces optimal power output values better than those produced by applying the coarse sampling/gradient ascent method and at a fraction of the computation time. However, Bates' GA settings were found to be incorrectly tuned for the extended GA runs. With increased mutation rate and extending to 50 generations, incremental improvement in cell output power was obtained for almost every cell configuration but with diminishing returns on computational time and resources.

The entire multi-junction solar cell was modeled using a real-valued GA implementation. The results from exercising the model were inconclusive due to identical power values derived from different cell configurations; the root cause needs to be investigated. One step that may help in the investigation is to develop a working model of the tunnel junction in Silvaco ATLAS^{TM}, allowing for the direct measurement of multi-junction cell output without the need to either conduct interpolation with the attendant problem of introducing approximation errors. A more computation-intensive method would be to run the iterative current matching routine on a multi-junction cell while using a 50 or more generation genetic algorithm to optimize individual junction performance at each target thickness level.

The developers and maintainers of LAM MPI have pooled their efforts with other MPI projects to create a new distributed processing environment called Open MPI. Open MPI combines the best features of the various MPI

implementations. Adaptation of the distributed computing code used in this thesis to this new system would likely not be that difficult. The benefit of doing so would be to have a more robust system on which to run experiments. Another method of distributing the work would be to develop the experimental apparatus in the new MATLAB™ Distributed Computing Engine and Toolbox, leveraging its ease of use, pre-built functions and tutorials, and compatibility with Windows.

Some additional avenues for further research are reviewing material property parameters and tuning the model for working under realistic operational parameters for space. Michalopoulos and Green each conducted a literature search for experimental data on the materials used in these multi-junction solar cells. Since many of the materials are fairly new in solar cells, there needs to be periodic reviews for updated data in order to improve the accuracy and performance of cell models within the Silvaco software. The users of Silvaco models have the ability to change environmental constraints such as temperature. Modification of the models to more closely resemble actual operating conditions (e.g., the more extreme thermal cycling in space over an extended period of time) would improve the utility of the models and provide more data for verification against documented power output levels.

APPENDIX A DETAILED REVIEW OF AN INPUT DECK

For the sake of brevity, a single junction GaAs input deck is used for this illustration. The first command tells DeckBuild™ that it will need to use the ATLAS™ device simulator for this simulation.

```
go atlas
```

The next section does not actually build the model. Instead, it is merely defining variables to be used later in the simulation. The variables from windowThick down through bsfDop are the variables which were optimized as discussed earlier in this thesis. In reading, note that actual values are simply written but any references to a previously defined variable include a dollar sign ($) at the beginning of the referenced variable name.

```
go atlas
set cellWidth=500
set capWidthpercent=8
set divs=10
set contThick=0.1
set capThick=0.3
set capDop=1e20

# The following 8 lines are the variables being optimized.
set windowThick=0.01
set winDop=2.15e17
set emitterThick=0.01
set emitDop=1e16
set baseThick=3.19467
set baseDop=1e16
set bsfThick=0.03533
set bsfDop=2.15e19
# This is the end of variables being optimized.

set cellWidthDiv=$cellWidth/$divs
set width3d=100e6/$cellWidth
set capWidth=0.01*$capWidthpercent*$cellWidth/2
set capWidthDiv=$capWidth/($divs/2)
set cellWidthHalf=$cellWidth/2
set bsfLo=0
set bsfHi=$bsfLo-$bsfThick
set bsfDiv=$bsfThick/$divs
set baseLo=$bsfHi
set baseHi=$baseLo-$baseThick
set baseMid=$baseLo-$baseThick/2
set baseDiv=$baseThick/$divs
set emitterLo=$baseHi
set emitterHi=$emitterLo-$emitterThick
set emitterDiv=$emitterThick/$divs
set windowLo=$emitterHi
```

```
set windowHi=$windowLo-$windowThick
set windowDiv=$windowThick/$divs
set capLo=$windowHi
set capHi=$capLo-$capThick
set contLo=$capHi
set contHi=$contLo-$contThick
set contDiv=$contThick/$divs
set lightY=$emitterHi-5
```

As described, a virtual mesh is defined throughout the volume of the device to be simulated. Every intersection between mesh lines is where differential equations are applied to determine device performance. Therefore, the specification of the mesh is tuned to the type of device being simulated. The mesh is much more fine around the intersection between cell layers and in regions where the majority of electrical activity takes place. Seemingly minor changes in the mesh can cause large changes in output values and even cause simulations to fail. For readability, ATLAS™ uses the # character to denote comments. On some lines, a double "##" operator is used. This is purely programmer discretion and done to denote section headings for readability. As long as there is a single #, the DeckBuild™ application will ignore the text for the rest of the line.

```
mesh width=$width3d
## X-Mesh
x.mesh loc=-$cellWidthHalf spac=$cellWidthDiv
x.mesh loc=-$capWidth spac=$capWidthDiv
x.mesh loc=$capWidth spac=$capWidthDiv
x.mesh loc=$cellWidthHalf spac=$cellWidthDiv
## Y-Mesh
# Top contact
y.mesh loc=$contHi spac=0
y.mesh loc=$contLo spac=0
# Cap
# Window
y.mesh loc=$windowHi spac=$windowDiv
y.mesh loc=$windowLo spac=$windowDiv
# Emitter
y.mesh loc=$emitterLo spac=$emitterDiv
# Base
y.mesh loc=$baseMid spac=$baseDiv
# BSF
y.mesh loc=$bsfHi spac=$bsfDiv
y.mesh loc=$bsfLo spac=$bsfDiv
```

The following section begins to define the actual structure of the cell. Each line defines a type of material to be used as well as the position and

dimensions of the region within the cell. Some comments are included to show how to adapt the input deck to simulate different types of cells.

```
#####################################
## CURRENTLY SET UP FOR: GaAs CELL ##
#####################################
## Regions [for InGaP cell, change region 1 to GaAs (v. Vacuum)
## and remove region 8 (bogus contact)]
## [for all others, change materials only]
# Cap
region num=8 material=Vacuum x.min=-$capWidth x.max=$capWidth y.min=$contHi
y.max=$contLo
region num=1 material=Vacuum x.min=-$capWidth x.max=$capWidth y.min=$capHi
y.max=$capLo
region num=2 material=Vacuum x.min=-$cellWidthHalf x.max=-$capWidth
y.min=$contHi y.max=$capLo
region num=3 material=Vacuum x.min=$capWidth x.max=$cellWidthHalf y.min=$contHi
y.max=$capLo
# Window [for Ge cell, use AlGaAs with x.comp=0.2]
region num=4 material=InGaP x.min=-$cellWidthHalf x.max=$cellWidthHalf
y.min=$windowHi y.max=$windowLo
# Emitter
region num=5 material=GaAs x.min=-$cellWidthHalf x.max=$cellWidthHalf
y.min=$emitterHi y.max=$emitterLo
# Base
region num=6 material=GaAs x.min=-$cellWidthHalf x.max=$cellWidthHalf
y.min=$baseHi y.max=$baseLo
# BSF
region num=7 material=InGaP x.min=-$cellWidthHalf x.max=$cellWidthHalf
y.min=$bsfHi y.max=$bsfLo
## Electrodes [for InGaP cell, add cathode (gold) and remove
cathode(conductor)]
#electrode name=cathode material=Gold x.min=-$capWidth x.max=$capWidth
y.min=$contHi y.max=$contLo
electrode name=cathode x.min=-$cellWidthHalf x.max=$cellWidthHalf
y.min=$windowHi y.max=$windowHi
electrode name=anode x.min=-$cellWidthHalf x.max=$cellWidthHalf y.min=$bsfLo
y.max=$bsfLo
```

The next section outlines the doping to be used for each region. All the concentrations are based on variables defined at the beginning of the input deck.

```
## Doping [for InGaP cell, uncomment cap doping]
# Cap
#doping uniform region=1 n.type conc=$capDop
# Window
doping uniform region=4 n.type conc=$winDop
# Emitter
doping uniform region=5 n.type conc=$emitDop
# Base
doping uniform region=6 p.type conc=$baseDop
# BSF
doping uniform region=7 p.type conc=$bsfDop
```

The following section defines the actual material properties for the various materials used in the cell. Silvaco has built-in libraries for a large number of

materials. However, many of the materials used in solar cell fabrication are rare and not frequently used in the semiconductor devices commonly modeled in the Silvaco TCAD suite. When the properties are not specifically defined or in the software library, ATLAS™ will revert to default values. When this happens, results are drastically affected. The materials are explicitly defined to avoid this predicament. The values used are based on previous research conducted at NPS [Refs. 10,11]. The numbers are partly from published literature and partly from calculation based on reasonable assumptions. Although solar cell manufacturers probably have more detailed and tested data, it is of commercial value to them to protect it as proprietary information. Therefore, it is worth periodic review and validation to adjust the accuracy of these numbers. The \ operator seen at the end of some lines is instructs ATLAS™ that the following line belongs with the current line but has been separated for readability purposes. All files referenced need to be with the input deck in the default working directory for the simulation to proceed without error.

```
## Material properties
# Opaque contact [comment out for InGaP cell]
material region=8 real.index=1.2 imag.index=1.8
# Vacuum (for zero reflection) [change to match window material (InGaP use
Vacuum_AlInP)]
# [for InGaP cell, comment out region 1]
material region=1 index.file=Vacuum_InGaP.opt
material region=2 index.file=Vacuum_InGaP.opt
material region=3 index.file=Vacuum_InGaP.opt
# GaAs
material material=GaAs EG300=1.424 PERMITTIVITY=12.9 AFFINITY=4.07 \
  NC300=4.7E17 NV300=9E18 INDEX.FILE=GaAs.opt COPT=7.2E-10 \
  AUGN=5E-30 AUGP=1E-31
# InGaP
material material=InGaP EG300=1.9 PERMITTIVITY=11.62 AFFINITY=4.16 \
  NC300=1.3E20 NV300=1.28E19 index.file=InGaP.opt COPT=1E-10 \
  MUN=4000 MUP=200 AUGN=3e-30 AUGP=3E-30
# Ge
material material=Ge EG300=0.661 PERMITTIVITY=16.2 AFFINITY=4 \
  NC300=1E19 NV300=5E18 index.file=Ge.opt COPT=6.41E-14 \
  MUN=3900 MUP=1900 AUGN=1E-30 AUGP=1E-30
# AlGaAs
material material=AlGaAs MUN=9000 MUP=100 INDEX.FILE=AlGaAs.opt
# AlInP (=InAsP)
material material=InAsP EG300=2.4 PERMITTIVITY=11.7 AFFINITY=4.2 \
  NC300=1.08E20 NV300=1.28E19 index.file=AlInP.opt COPT=1.2E-10 \
  MUN=2291 MUP=142 AUGN=9E-31 AUGP=9E-31
# AlInGaP (=InAlAsP)
material material=InAlAsP EG300=2.4 PERMITTIVITY=11.7 AFFINITY=4.2 \
  NC300=1.2E20 NV300=1.28E19 index.file=AlInP.opt COPT=1E-10 \
  MUN=2150 MUP=141 AUGN=3e-30 AUGP=3E-30
```

```
# InGaNAs
material material=InGaNAs EG300=1.0 PERMITTIVITY=11.7 AFFINITY=4.05 \
  NC300=3.2e19 NV300=1.8e19 index.file=InGaNAs.opt COPT=7.2e-10 \
  MUN=3000 MUP=150
# Gold
material material=Gold real.index=1.2 imag.index=1.8
```

The following section allows the user to specify the mathematical models to be applied to the various regions within the cell. CONMOB is a concentration dependent electron mobility model for GaAs and Si [Ref. 9]. OPTR specifies the band to band recombination model and print simply instructs ATLAS™ to add the recombination data to the log file generated at runtime [Ref. 9].

```
## Models [InGaP cell, 1; GaAs cell, 5&6; InGaNAs cell, 7]
#models region=1 CONMOB
models region=5 CONMOB
models region=6 CONMOB
#models region=7 CONMOB
models OPTR print
```

The Light beams section is where the user may define what light spectrum, intensity, and angle to shine on the solar cell. This feature allowed Bates to optimize cell designs for the light spectrum in both earth orbit (AM0) and on Mars [Ref. 7]. The struct command following the light beams defines a structure file with all the physical setup information for the cell. The commented out tonyplot command generates a graphical picture of the cell model.

```
## Light beams
beam num=1 x.origin=0 y.origin=$lightY angle=90 back.refl \
  power.file=AM0nrel.spec \
  wavel.start=0.12 wavel.end=2.4 wavel.num=50
#struct outfile=SingleCell_webf.str
#tonyplot SingleCell_webf.str
```

The following section of the input deck begins the collection data. The first section exposes the cell to light and then extracts the short circuit current from the data file. Various current values along the expected current-voltage curve are then calculated and defined based on fractions of the short circuit current.

```
solve init
method gummel newton maxtraps=10 itlimit=25
solve b1=0.9
log outfile=CHR10485774.log
solve b1=0.95
log off
extract init infile="CHR10485774.log"
extract name="isc" max(i."cathode")
```

```
## set isc=$isc*$width3d
set isc=$isc
set i1=$isc/10
set i2=$i1+$isc/10
set i3=$i2+$isc/10
set i4=$i3+$isc/10
set i5=$i4+$isc/10
set i6=$i5+$isc/20
set i7=$i6+$isc/20
set i8=$i7+$isc/20
set i9=$i8+$isc/20
set i10=$i9+$isc/20
set i11=$i10+$isc/40
set i12=$i11+$isc/40
set i13=$i12+$isc/40
set i14=$i13+$isc/40
set i15=$i14+$isc/40
set i16=$i15+$isc/80
set i17=$i16+$isc/80
set i18=$i17+$isc/80
set i19=$i18+$isc/80
set i20=$i19+$isc/80
set i21=$i20+$isc/80
set i22=$i21+$isc/80
set i23=$i22+$isc/80
set i24=$i23+$isc/80
set i25=$i24+$isc/80-0.00001
```

The final code section is where the actual IV curve measurements are made based on the current values defined in the previous section. Per the comments, solve points are tailored to expected max power ranges for each type of cell in order to minimize computation time. The final lines close out the log file and create a done signal for MATLAB™. The commented tonyplot command, if uncommented, generates a graphical IV curve plot based on simulation results.

```
log outfile=CHR10485774.log
method newton maxtraps=10 itlimit=100
solve b1=0.95
contact name=anode current
method newton maxtraps=10 itlimit=100
## Pmax points [InGaP 18-25; GaAs 15-25; InGaNAs 13-25; Ge 11-25]
solve ianode=-$i25 b1=0.95
solve ianode=-$i24 b1=0.95
solve ianode=-$i23 b1=0.95
solve ianode=-$i22 b1=0.95
solve ianode=-$i21 b1=0.95
solve ianode=-$i20 b1=0.95
solve ianode=-$i19 b1=0.95
solve ianode=-$i18 b1=0.95
solve ianode=-$i17 b1=0.95
solve ianode=-$i16 b1=0.95
solve ianode=-$i15 b1=0.95
solve ianode=-$i14 b1=0.95
solve ianode=-$i13 b1=0.95
solve ianode=-$i12 b1=0.95
solve ianode=-$i11 b1=0.95
```

```
##now solve for Voc
solve ianode=0 b1=0.95
log off
## Full I-V curve plot
#tonyplot SingleCell_webf.log -set pmax.set
##
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: CHALLENGES IN ADAPTING WINDOWS ATLAS™ INPUT DECKS TO ATLAS™ UNDER UNIX

While all previous research had been conducted using Silvaco ATLAS™ and MATLAB™ under the Windows operating system, running the same input decks under ATLAS™ running on the Linux operating system proved initially challenging. In fact, the input decks for Windows using DeckBuild™ will not execute correctly under Linux without modification.

The first problem encountered was that the simulation runs would fail with a variety of odd errors reported. Upon consultation with Silvaco engineers, the problem was identified as a text formatting difference between Windows and Linux. Since the input decks of initial simulation attempts were based on copying text from a previous thesis and pasting in a text editor in Windows, some odd characters were introduced. Later, when Bates original files were obtained, they simulated without this problem. Fortunately, Linux has a built-in utility for correcting this problem. The utility, dos2unix, is run from the command prompt in a terminal window. Although it has many options and capabilities, the defaults worked correctly in this situation. As an example, consider the Windows file Chr0.in which requires conversion. The user first moves into the Linux directory in which the file is located and executes the following command:

Prompt> dos2unix ./CHR0.in <return>

The next problem encountered was a difference in how the results of computation are automatically stored and referenced using Silvaco tools under Windows and Linux. The solve portion of an input deck first solves for short circuit current, extracts this value, and then sets up all the current-voltage curve points to solve for based on fractions of the short circuit current. Under Windows, values are automatically saved to a file and then that file is automatically referenced for extraction. Under Linux, this log file must first be explicitly established before solving for the current, and then closed before being referenced. The extract init statement must be told the log file's name. In

addition, the current value extracted under Windows is only for 1/200000th of the mesh and must be multiplied. Under Linux, the value extracted is the current for the entire cell and thus does not need to be multiplied by the mesh size. When the original input deck was run, a pop up window gave the following announcement: "** INFORMATION ** Monitor String from selected list detected. The simulation has been stopped." In the log file from the simulation, the following comments were recorded before the simulation halted:

```
Warning: 'set' syntax not recognized.
Deckbuild passing 'set' command to simulator.
set isc=*200000
 ** ERROR #  1 **
 * Invalid card type specification *
 ==> set
```

Below are sample sections of input decks. The Linux version has been adjusted to simulate correctly.

solve init method gummel newton maxtraps=10 itlimit=25 solve b1=0.9 ## Getting Isc for I-V curve points method newton maxtraps=10 itlimit=100 solve b1=0.95 extract name="isc" max(i."cathode") set isc=$isc*$width3d set i1=$isc/10 set i2=$i1+$isc/10	solve init method gummel newton maxtraps=10 itlimit=25 solve b1=0.9 log outfile=CHR4294967295.log solve b1=0.95 log off extract init infile="CHR4294967295.log" extract name="isc" max(i."cathode") ## set isc=$isc*$width3d set isc=$isc set i1=$isc/10 set i2=$i1+$isc/10

Table 6. Sample Extraction code under Windows (Left) and Linux (Right)

Upon successful execution of an adapted Windows input deck under Linux, cell output data was compared. Although current-voltage curves were similar, they were not identical. Upon inspection of the log files automatically generated by ATLAS^{TM}, the following differences were observed:

ATLAS> solve init CONSTANTS: Boltzmann's constant = 1.38066e-023 J/K Elementary charge = 1.6023e-019 C Permitivity in vacuum = 8.85418e-014 F/cm	ATLAS> solve init CONSTANTS: Boltzmann's constant = 1.38066e-23 J/K Elementary charge = 1.60219e-19 C Permitivity in vacuum = 8.85419e-14 F/cm

Temperature = 300 K Thermal voltage = 0.0258502 V	Temperature = 300 K Thermal voltage = 0.025852 V

Table 7. Scientific Constants used by ATLAS™ under Windows (Left) and Linux (Right)

To ensure results are consistent with previous work, ten input decks were run using the original Windows format and under Linux using the adaptation described above. The output values obtained under Windows and Linux were then compared. The input decks chosen were a wide range intended to ensure every variable has at least one parameter change. In the plot below, nine of the chromosome pairs' iv plots are displayed. One chromosome had an order of magnitude lower power and is not shown on this plot.

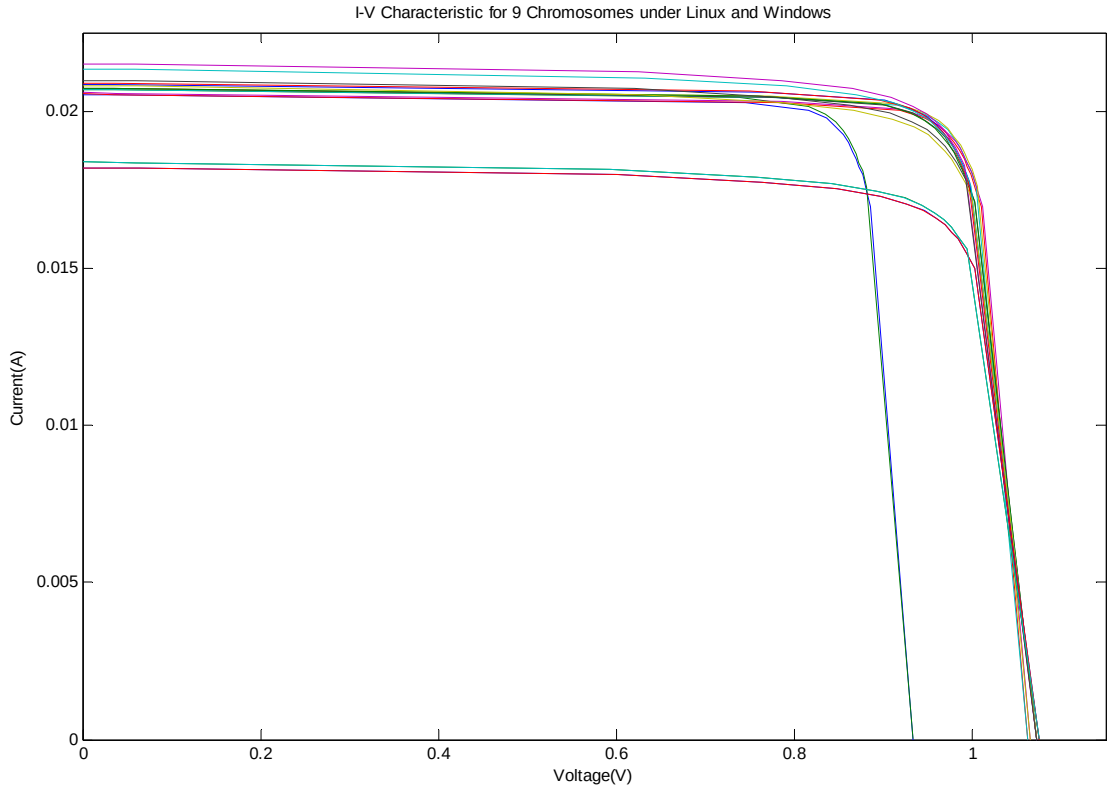


Figure 26. Comparison of IV characteristics obtained under Windows and Linux

When plotted individually, it is easy to see that each pair (Windows and Linux for the same chromosome) follow almost exactly the same plot. MATLAB™ analysis revealed that the largest percentage difference in obtained

power values was around 5% - and this difference was observed on only 5 out of 150 current voltage points. The majority were less than one percent. Both methods produced the same ranking of chromosomes according to maximum power. It is clear that both versions of input decks are effectively defining the cell configuration and obtaining results. In addition, the absolute value of the results obtained is reasonably close.

Engineers at Silvaco have explained the discrepancy by noting that there are differences in the way Windows and Linux handle floating point number formats and arithmetic. Their recommendation was that if the absolute value of the number is critical (as opposed to the relative ranking) then the Linux or Unix generated values are likely more accurate.

APPENDIX C: DISTRIBUTED COMPUTING PROGRAMMER'S NOTES

Before starting this thesis work, the author had two undergraduate programming classes (one in ADA, the other in C) and one graduate class with simplistic C programs. The majority of software problems encountered were easily solved with a fairly basic set of programming skills. The LAM-MPI web site had a very thorough self paced class on the basics of programming for LAM-MPI distributed computing. In addition, the leaders of LAM-MPI along with other MPI projects have now consolidated their efforts to build Open MPI which was coded from scratch to incorporate the best of all the MPI projects. Exact duplication of the methods used are not likely to work on other platforms and might not be advisable even if they did. When originally conceived and coded, the software used for this thesis had many provisions for capabilities which, at the time, were thought to be useful to this research. The test of time has shown that simple is better, and re-coding of significant portions has been undertaken just to make the code simpler and easier to maintain. The code base has gone through several generations as the research progressed. The coarse sampling, gradient, remote genetic algorithm, and remote real-valued multi-junction optimization all required adjustments and changes to the code. As such, the goal of this appendix is not to disclose at length all the code used, but simply to highlight a couple of the more difficult challenges in developing the distributed computing software.

The first major obstacle was that Silvaco ATLAS™ does not always give predictable results. When Bates used MATLAB™ to make system calls to run ATLAS™, ATLAS™ crashes or unexpected results would often cause the MATLAB™ code to halt or crash when it tried to deal with the problems. Upon further investigation, updated versions of Windows and MATLAB™ had a slightly different system call function. In previous versions the system call would return to MATLAB™ some information as to the status of the system call. Bates then had MATLAB™ sit in a waiting loop looking for a file called done.log. The last

item of his ATLAS™ input deck instructed ATLAS™ to create an empty log file called done.log. However, ATLAS™ doesn't always complete a simulation run. Bad data or a corrupted input deck can cause the simulation to fail. The changes in system calls made this situation difficult to detect from the MATLAB™ environment. In the course of this thesis, no method of checking up on the simulations was found within the MATLAB™ environment.

Since the distributed computing platform used in this thesis was running on LAM-MPI on the Linux operating system, other methods of process supervision were available. The client process, when instructed to conduct a simulation, would utilize the C fork and execl commands to accomplish process supervision. This implementation is well articulated in the documentation header for the code below:

```
/*
 *          Created on Oct 11, 2004 4:02:33 PM
*/
/*
 *                       MPI based Solar Cell Simulation
 *
 *
 *    Created:          11 Oct 2004
 *    Last Modified:    18 Nov 2005
 *    Author:           James Utsler
 *

 *  mpi_solar_atlas_call.c contains the source code for calling atlas
 *  to act on a specified input file or "deck" under the silvaco TCAD
 *  tools.  With minor modification, this may be used to call a
 *  different application.  The general concept is the need to call
 *  atlas but also monitor
 *  the "atlas call" in case it freezes, hangs, or crashes in some way.
 *  To accomplish this, the unix fork and execl commands are used to
 *  fork program execution and then have the child program call atlas.
 *  The parent monitors the child process by looking at the child
 *  process's status file which is located in the /proc/procid#/status
 *  file, where procid# is replaced by the actual process id of the
 *  child.  A specific position in the status file gives the state of
 *  execution.  This program only checks if the character is a Z which
 *  denotes zombie status.  Zombie status means that the child has
 *  completed execution and is waiting for the parent to "reap" them.
 *  If in this condition, the parent "reaps" the child and then returns
 *  to the calling program stating successful execution.  If the child
 *  process is not in the zombie state, the parent goes to sleep for a
 *  specified period of time, wakes up, and checks the childs status
 *  again.  This continues until a "timed-out" threshold is met.  The
 *  interval between checks and timed-out threshold are specified in
```

```
 *  the call to this program.
 *
 *  Inputs:
 *    chromosome - specifies what chromosome to call atlas on, this
 *       program uses chromosome to build the input filename and log
 *       filename for the atlas call
 *    how_long_to_wait - specifies the length of time the parent
 *       process waits before "killing" the child and reporting faulty
 *       execution (time in seconds)
 *    how_long_between_checks - specifies how long the parent sleeps
 *       between checks on the child (time is in seconds) before
 *       calling this program, the master program has already written
 *       the input file which corresponds to this chromosome
 *
 *  Returns:
 *    0 with successful execution.
 *    1 with timed-out or a faulty system call
 *
 */

#include <stdio.h>
#include <unistd.h>

int supervised_atlas_call(unsigned long int chromosome,
   int how_long_to_wait,
int how_long_between_checks)
{

   int pid;

   FILE *process_status_file;
   int i=0;
   int return_value;
   char chromstr[20];
   char cmdstr[40]="kill -TERM \0";
   char pidstr[10];
   char tempstr[50]="/proc/\0";

   char status;

   printf("Simulating Chrom %lu\n",chromosome);
   pid=fork();

   if (pid!=0) {
      /*printf("Parent:child's pid=%d\n",pid);*/
      sprintf(cmdstr,"kill -TERM %i\0",pid);  /* this builds the kill
                                                 signal command string */
      sprintf(pidstr,"%lu\0",pid);

      sprintf(tempstr,"/proc/%lu/status\0",pid);

      while (1) {
         sleep(how_long_between_checks);
         i=i+how_long_between_checks;     /* increment time counter */
         process_status_file=fopen(tempstr,"r");  /* open the status
```

```
                                                    file */

         if (process_status_file!=NULL) {         /* if the file was
                                                    "Openable" */
            fscanf(process_status_file,"%*s %*s\n%*s %c",&status);
               /* pick out the status letter */

            /*printf("Parent:child status= %c\n",status);*/
            fclose(process_status_file);           /* close the file */
         }
         else break;    /* exit the loop if file was not accessible */
         if (status=='Z') {     /* exit the loop if the child completed
                                   processing */
            return_value=0;     /*  it would have status Z -> Zombie */
            wait(pid);
            break;
         }


         if (i>how_long_to_wait) {     /* kill child if time is over
                                  user defined seconds per loop */
            /*printf("parent killing child\n");*/
            system(cmdstr);
            return_value=1;
            break;
         }
         /*printf("parent waiting time=%i\n",i);*/
      }
      /*printf("Parent:exiting now\n");*/
   }
   else {
      /*printf("Child executing program:my pid=%d\n",pid);*/
      sprintf(chromstr,"CHR%lu.in\0",chromosome);
      sprintf(tempstr,"CHR%lu.txt\0",chromosome);
      /*execl("/bin/ping","ping","-c","1","www.yahoo.com",0);*/
      execl("/opt/silvaco2/bin/deckbuild","deckbuild","-run",
           "-ascii",chromstr,"-outfile",tempstr,0);
      printf("Problem with Atlas child process call!\n"); /* this line
                                             should not be executed */
      return_value=1;
   }
   return return_value;
}
```

Once the above code completes, assuming a successful simulation, the client program parses out the ATLAS™ output file to ensure that sufficient data was generated and then reports back as to the success of the simulation. If the ATLAS™ process fails or the data does not check out, the distributed system monitor is notified and the failed simulation is noted in status files.

Another concern with process completion is that the processors of the distributed computing platform are located in a student lab. In some cases students would reboot a computer if something went awry or occasionally shut down a computer at the end of their work thinking it was an environmentally appropriate step to conserve energy. LAM-MPI is not flexible on adding or deleting nodes after the platform is initiated. In early software versions of this application, this meant that the entire system might break down if the distributed system monitor was blocked waiting to send or receive a message from one of the clients. In latter versions of the software, a periodic check-in system was used. Clients always default to sending a non-blocking message to the distributed system monitor asking for work. If no work is available, the client is simply told to check in at a later time. However, every time this occurs, the distributed system monitor scrubs its records to see if any simulation has exceeded a maximum time threshold. If this is the case, the simulation would be assigned to another computer and the timed-out client would be black-listed so that no further results would be accepted from that client. This also prevents the case of two nodes reporting results for the same simulation.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D: MATLAB™ GENETIC ALGORITHM AND DIRECT SEARCH TOOLBOX NOTES

Working within the Genetic Algorithm and Direct Search Toolbox made modifying GA parameters significantly easier. It allowed the programmer to focus more on the research being conducted. The GUI allows the easy configuration of parameters and offers an option to generate a MATLAB™ script which will execute the parameters chosen without the GUI interface. For consistency among multiple simulations, the scripts were used for the actual simulations. The following code was used for the extended GA runs on single junction cells:

```
function [X,FVAL,REASON,OUTPUT,POPULATION,SCORES] =  ga1
%%   This is an auto generated M file to do optimization with the
Genetic Algorithm and
%    Direct Search Toolbox. Use GAOPTIMSET for default GA options
structure.

generation=0;
save('generation.txt','generation','-ascii');
clear generation;

FID=fopen('path_list.txt','w');
fprintf(FID,'./Ge_320.00 25 11 4 320.00');
fclose(FID);

system('copy path_list.txt Z:');

%%Fitness function
fitnessFunction = @single_junction_remote_batch_fitness_function;
%%Number of Variables
nvars = 28;
%Linear inequality constraints
Aineq = [];
Bineq = [];
%Linear equality constraints
Aeq = [];
Beq = [];
%Bounds
LB = [];
UB = [];
%Nonlinear constraints
nonlconFunction = [];
%Start with default options
options = gaoptimset;
%%Modify some parameters
options = gaoptimset(options,'PopulationType' ,'bitString');
```

```
options = gaoptimset(options,'PopulationSize' ,36);
options = gaoptimset(options,'Generations' ,50);
options = gaoptimset(options,'StallTimeLimit' ,Inf);
options = gaoptimset(options,'SelectionFcn' ,@selectionroulette);
options = gaoptimset(options,'CrossoverFcn' ,@crossovertwopoint);
options = gaoptimset(options,'MutationFcn' ,{ @mutationuniform 0.10 });
options = gaoptimset(options,'Display' ,'off');
options = gaoptimset(options,'PlotFcns' ,{ @gaplotbestf
@gaplotscorediversity });
options = gaoptimset(options,'Vectorized' ,'on');
%%Run GA
[X,FVAL,REASON,OUTPUT,POPULATION,SCORES] =
ga(fitnessFunction,nvars,Aineq,Bineq,Aeq,Beq,LB,UB,nonlconFunction,opti
ons);
```

The following code is the fitness function used for single-junction extended GA runs.

```
function
fitness_values=InGaP_single_junction_remote_batch_fitness_function(gene
ration_trait_vector)
%
% This function takes a matrix of trait values for each genetic
algorithm
% generation and acts as a liason between the genetic algorithm running
on
% a windows machine and a distributed computing system which will
handle
% all solar cell simulation.  The communication is acheived using
simple
% status bit files and data files.  When ready to simulate a
generation,
% this function will save the data in a formatted manner to the shared
file
% system of the distributed computing project.  It will then set a file
% with a status bit indicating that the data is ready.  The distributed
% system will then simulate all the data, archive the input and output
files,
% and then update a status file indicating that results are ready.  The
% data will then be formatted for return to the MATLAB GA tool and this
% function will exit.  It will be called again at the beginning of the
next
% iteration.
%
%
% clear

format long;

%% First convert the bit strings to chromosome numbers compatible with
% previously built GA/distributed computing code
%
```

```
[number_chroms,chrom_bitlength]=size(generation_trait_vector)
FID=fopen('master_chromosome_list.txt','w');
chromosomes=[];
for i=1:number_chroms

chromosomes=[chromosomes;bits_to_traits(generation_trait_vector(i,:),28
)];
    fprintf(FID,'%d\n',chromosomes(i));
end
fclose(FID);

system('copy master_chromosome_list.txt Z:');

results_ready=0;
save('Z:\results_ready.txt','results_ready','-ascii');

chromosomes

%% Next, set the status bit to let the remote system know that the data
% file is ready for simulation.

sim_start=1;
save('Z:\simulation_data_ready.txt','sim_start','-ascii');

%% Now periodically check the status bit which indicates that
simulation is
% complete.

results_ready=0;
while results_ready==0
     results_ready=load('Z:\results_ready.txt');
end

%% Now create a directory to store the resulting log files
generation=load('generation.txt');
command=sprintf('copy Z:\\temp_result_matrix.txt
C:\\GenAlg\\Gen%d.txt',generation);
system(command);


command=sprintf('C:\\GenAlg\\Gen%d.txt',generation);
simulation_data=load(command)
fitness_values=sort_fitness_results(chromosomes,simulation_data);
generation=generation+1;
save('generation.txt','generation','-ascii');
system('erase Z:\temp_result_matrix.txt');
return
```

For single-junction optimization, the data validation was conducted within the distributed computing system. When simulation runs were complete, a summary file giving a line of information for each chromosome was passed back

to MATLAB™. The following function was used to match up the returned data file to the known chromosome list in MATLAB™ to ensure that fitness values are returned in the correct format.

```
function
ordered_fitness=sort_fitness_results(given_population,computed_results)
%
%  When data comes back from a "vectorized" distributed computing
%  simulation, there is the possibility that a specific chromosome
%  would not simulate successfully.  In this case, there will be no
%  result entry coming back from the distributed computing setup for
%  that specific chromosome.  This function compares the result file to
%  the actual population and ensures all fitness values correspond to
%  the correct chromosomes.  In the case where no result comes back, it
%  is assigned the fitness value of 0.
%
%
given_elements=length(given_population);
result_elements=size(computed_results,1);
given_population(:,2)=1:given_elements;

given_population=sortrows(given_population,1);
computed_results=sortrows(computed_results,1);

result_placeholder=1;
ordered_fitness=[];
for i=1:given_elements
    updated=0;
    for j=1:result_elements
        if given_population(i,1)==computed_results(j,1)

ordered_fitness=[ordered_fitness;given_population(i,2),computed_results
(j,2)];
            updated=1;
        end
    end
    if updated==0
        ordered_fitness=[ordered_fitness;given_population(i,2),0];
    end
end
ordered_fitness=sortrows(ordered_fitness,1);
ordered_fitness=-ordered_fitness(:,2);
```

When conducting the real-valued optimization of multi-junction cells, the author learned that not all features of the toolbox are fully implemented. When using real-valued number representation of traits, the toolbox allows the user to specify upper and lower bounds for trait values. In addition, both linear and nonlinear constraints relating variables may be introduced. However, when these options are used, the toolbox defaults to a condition in which it's initial

population is a single test chromosome. When complete, the second generation consists of clones of the test chromosome with a few minor mutations. This essentially usurps the genetic diversity which is key in a genetic algorithm based optimization. Discussion with MATLAB™ engineers confirmed that this will be fixed in future versions and that there are ways to work around this problem. As a simple fix, the method used in this thesis is to manually code in MATLAB™ a routine to build a random population within the upper and lower bound constraints for each trait. The GA function of the toolbox is then called. The following code was used:

```
function [X,FVAL,REASON,OUTPUT,POPULATION,SCORES] =
ga2owncreate(population)
%%   This is an auto generated M file to do optimization with the
Genetic Algorithm and
%    Direct Search Toolbox. Use GAOPTIMSET for default GA options
structure.



generation=0;
save('generation.txt','generation','-ascii');
clear generation;
figure(1);

%%Fitness function
fitnessFunction = @quad_junction_remote_batch_fitness_function;
%%Number of Variables
nvars = 29;
%Linear inequality constraints
Aineq = [];
Bineq = [];
%Linear equality constraints
Aeq = [];
Beq = [];
%Bounds
%          window    emitter    base      bsf
ingaplb = [0 1e16    0 1e16     0 1e16    0 1e16];
%          window    emitter    base      bsf
ingapub = [10 1e20  10 1e20   10 1e20  10 1e20];
%          window    emitter    base      bsf
gaaslb = [0 1e16    0 1e16     0 1e16    0 1e16];
%          window    emitter    base      bsf
gaasub = [10 1e20  10 1e20   10 1e20  10 1e20];
%          window    emitter    base      bsf
inganaslb = [0 1e16    0 1e16     0 1e16    0 1e16];
%          window    emitter    base      bsf
inganasub = [10 1e20  10 1e20   10 1e20  10 1e20];
%          window    emitter    base
```

```
gelb =     [0 1e16    0 1e16    1e16];
%          window    emitter    base
geub =     [10 1e20  10 1e20   1e20];

LB=[ingaplb gaaslb inganaslb gelb];
UB=[ingapub gaasub inganasub geub];

% now create the initial population by multiplying a matrix of random
% variables by a scale factor matrix (the difference in upper and lower
% bounds) and then adding the lower bound to the scaled values

popsize=90;
population=rand(popsize,nvars);
scalar_factor=UB-LB;
for i=1:popsize
    population(i,:)=scalar_factor.*population(i,:);
    population(i,:)=LB+population(i,:);
end

drews_data=load('drews_data.txt')
population=[drews_data;drews_data;drews_data;population(7:popsize,:)]
% LB=[];
% UB =[];
%Nonlinear constraints
nonlconFunction = [];
%Start with default options
options = gaoptimset;
%%Modify some parameters
options = gaoptimset(options,'InitialPop' ,population);
options = gaoptimset(options,'PopulationSize' ,popsize);
options = gaoptimset(options,'Generations' ,Inf);
options = gaoptimset(options,'StallGenLimit' ,100);
options = gaoptimset(options,'StallTimeLimit' ,Inf);
options = gaoptimset(options,'TolFun' ,0);
options = gaoptimset(options,'TolCon' ,0);
options = gaoptimset(options,'SelectionFcn' ,@selectionroulette);
options = gaoptimset(options,'CrossoverFcn' ,@crossovertwopoint);
options = gaoptimset(options,'MutationFcn' ,{ @mutationadaptfeasible
0.01 });
options = gaoptimset(options,'Display' ,'off');
options = gaoptimset(options,'PlotFcns' ,{ @gaplotbestf
@gaplotscorediversity });
options = gaoptimset(options,'Vectorized' ,'on');
%%Run GA
[X,FVAL,REASON,OUTPUT,POPULATION,SCORES] =
ga(fitnessFunction,nvars,Aineq,Bineq,Aeq,Beq,LB,UB,nonlconFunction,opti
ons);
```

The following code is the fitness function used for real-valued multi-junction optimization:

```
function
fitness_values=quad_junction_remote_batch_fitness_function(generation_t
rait_vector)
%
% This function takes a matrix of trait values for each genetic
% algorithm generation and acts as a liason between the genetic
% algorithm running on a windows machine and a distributed computing
% system which will handle all solar cell simulation.  The
% communication is acheived using simple status bit files and data
% files.  When ready to simulate a generation, this function will save
% the data in a formatted manner to the shared file system of the
% distributed computing project.  It will then set a file with a status
% bit indicating that the data is ready.  The distributed system will
% then simulate all the data, archive the input and output files, and
% then update a status file indicating that results are ready.  The
% data will then be formatted for return to the MATLAB GA tool and this
% function will exit.  It will be called again at the beginning of the
% next iteration.
%
%




format long;
%generation_size=36;
%% First save the data to a file for further simulation.  The file's
% format will be a chromosome number corresponding to the row in the
% trait matrix by the actual traits listed in order.
%
status_list=[1:size(generation_trait_vector,1)]'
%data_file=[chrom_numbers,generation_trait_vector]
generation_trait_vector

results_ready=0;
save('Z:\results_ready.txt','results_ready','-ascii');

save('Z:\generation_trait_data.txt','generation_trait_vector','-
ascii');
%save('C:\GenAlgeration_trait_data.txt','generation_trait_vector','-
ascii');

FID=fopen('master_chromosome_list.txt','w');
for i=1:size(generation_trait_vector,1)
    fprintf(FID,'%d\n',i);
end
fclose(FID);

system('copy master_chromosome_list.txt Z:');

%% Next, set the status bit to let the remote system know that the data
% file is ready for simulation.
```

```
sim_start=1;
save('Z:\simulation_data_ready.txt','sim_start','-ascii');

%% Now periodically check the status bit which indicates that
% simulation is complete.
generation=load('generation.txt');

%% Now create a directory to store the resulting log files

command=sprintf('mkdir C:\\GenAlg\\Gen%d',generation);
system(command);

results_ready=0;
while results_ready==0
    results_ready=load('Z:\results_ready.txt');
    command=sprintf('move Z:\\GenDataGood\\*.*
C:\\GenAlg\\Gen%d',generation);
    system(command);
    command=sprintf('move Z:\\GenDataBad\\*.*
C:\\GenAlg\\Gen%d',generation);
    system(command);
end

status_list=analyze_simulation_success(status_list)


system('erase Z:\completed_chromosome_logbak.txt');
system('erase Z:\chromosome_error_logbak.txt');

%% Now create a directory to store the resulting log files


command=sprintf('move Z:\\GenDataBad\\*.*
C:\\GenAlg\\Gen%d',generation);
system(command);
command=sprintf('move Z:\\GenDataGood\\*.*
C:\\GenAlg\\Gen%d',generation);
system(command);
command=sprintf('move Z:\\completed_chromosome_log.txt
C:\\GenAlg\\Gen%d',generation);
system(command);
command=sprintf('move Z:\\chromosome_error_log.txt
C:\\GenAlg\\Gen%d',generation);
system(command);
command=sprintf('move Z:\\generation_trait_data.txt
C:\\GenAlg\\Gen%d',generation);
system(command);

%% The following loop copies a log file to a local drive, calls a
% function to determine the cell's max power value, and then stores the
% result in a matrix listing chromosome numbers and the max power
% (fitness value).  Since MATLAB's GA Tool minimizes functions, the
% negative of the max power is stored.
```

```
save

for chrom=1:size(status_list,1)
    if status_list(chrom,2)==1
        filename=sprintf('C:\\GenAlg\\Gen%d\\CHR%d',generation,chrom);
        [isctot,voctot,imptot,vmptot,pmaxtot,fftot]=mj_ivmaxp(filename)
        status_list(chrom,3)=-pmaxtot;
    end
end

fitness_values=status_list(:,3)

%% Now format the data for return to the MATLAB genetic algorithm tool.
generation=generation+1;
save('generation.txt','generation','-ascii');
return
```

For multi-junction result interpretation, the raw data files were subjected to a data validation within the distributed computing system to ensure a simulation had completed. For output power calculation, the result files were interpreted from within MATLAB™ using a routine developed by Max Green in his thesis work. The following code was used:

```
function
[isctot,voctot,imptot,vmptot,pmaxtot,fftot]=mj_ivmaxp(runinfile)

format long;

datacol=textread([runinfile '.log'],'%*s%u%*[^\n]','headerlines',18);

numelect=datacol(1);
cols=datacol(2);

beams=mod(cols-4,numelect*3)+1;

beamstuff=[];
for i=1:beams
    beamstuff=[beamstuff '%*f'];
end

trodestuff=['%*f%*f%f%f%*f%*f']; pwredge=0;  badpmax=0;
for i=1:(numelect/2)
    [Io(:,i) Vo(:,i)]=textread([runinfile '.log'],['%*s' beamstuff
'%*f%*f%*f' trodestuff '%*[^\n]'], ...
        'headerlines',20);
    trodestuff=['%*f%*f%*f%*f%*f%*f' trodestuff];
    Po(:,i)=Io(:,i).*Vo(:,i);
    isc(i)=max(Io(:,i));
    [mincurrent indx]=min(Io(:,i));
    voc(i)=Vo(indx,i);
```

```
    [Pmax(i) indx]=max(Po(:,i));
    while Vo(indx,i)>Vo(indx+1,i)
        disp(['*** SUSPICIOUS PMAX' num2str(i) '=' num2str(Pmax(i)) '
DROPPED ***']);
        [Pmax(i) addon]=max(Po((indx+1):max(size(Po(:,i))),i));
        indx=indx+addon;
        badpmax=1;
    end
    if indx==2
        pwredge=1;
        disp(['*** INCOMPLETE LOWER BOUNDING OF PMAX' num2str(i) '
***']);
        numboundprob=numboundprob+1;
    elseif indx==(max(size(Po(:,i)))-1)
        pwredge=2;
        disp(['*** INCOMPLETE UPPER BOUNDING OF PMAX' num2str(i) '
***']);
        numboundprob=numboundprob+1;
    end
    FF(i)=Pmax(i)/(isc(i)*voc(i));
    imp(i)=Io(indx,i);
    vmp(i)=Vo(indx,i);
end

[pmaxtot,imptot,itotal,vtotal]=maxpower(Io,Vo,imp,isc,voc,numelect);

isctot=max(itotal);
voctot=max(vtotal);
vmptot=pmaxtot/imptot;
fftot=pmaxtot/(isctot*voctot);
Vtotmax=vtotal;
Iomax=Io;
Itotmax=itotal;

pmaxline=imptot*ones(size(vtotal));
xlim=1.1*max(vtotal);
ylim=1.1*max(isc);
figure(2);

if (numelect/2)==4

plot(Vo(:,1),Io(:,1),'b',Vo(:,2),Io(:,2),'r',Vo(:,3),Io(:,3),'g',...
        Vo(:,4),Io(:,4),'k',vtotal,itotal,'m',vtotal,pmaxline,'c:');
    legend('InGap','GaAs','InGaNAs','Ge','Total Cell',...
        ['Pmax= ' num2str(pmaxtot*1000) 'mW'],0);
elseif (numelect/2)==3

plot(Vo(:,1),Io(:,1),'b',Vo(:,2),Io(:,2),'r',Vo(:,3),Io(:,3),'g',...
        vtotal,itotal,'m',vtotal,pmaxline,'c:');
    legend('InGap','GaAs','Ge','Total Cell',...
        ['Pmax= ' num2str(pmaxtot*1000) 'mW'],0);
end

xlabel('Voltage (V)');
```

```
ylabel('Current (A)');
axis([0 xlim 0 ylim]);
figure(1);
```

The preceeding function relies on another power interpretation routine also developed by previous researchers at NPS. That routine follows:

```
function
[maxp,imaxp,itotal,vtotal]=maxpower(Io,Vo,imp,isc,voc,numelect)

% add up powers of each cell at lowest isc down to lowest imp
% record max power and overall imp

itry=linspace(min(imp),min(isc),10);
itry=[linspace(min(imp)*0.6,min(imp),10) itry];

% Io=known y's (decreasing)
% Vo=known x's (increasing)
% itry=given y's
% vtgt=target x's

istart(1)=2;
for i=1:(numelect/2)-1
    for j=istart(i):max(size(Io(:,i)))
        if Io(j,i)<0.00001
            istart(i+1)=j+1;
            break;
        end
    end
end
istart((numelect/2)+1)=max(size(Io(:,1)))+1;

for j=1:max(size(itry))
    maxpwr(j)=0;
    vtotal(j)=0;
    for i=1:(numelect/2)
        pivot=0;
        for x=istart(i):(istart(i+1)-1)
            if Io(x,i)<itry(j)
                pivot=x;
                if pivot==istart(i)
                    pivot=istart(i)+1;
                end
            end
            if pivot
                break;
            end
        end
        if ~pivot
            pivot=istart(i+1)-1;
        end
        linterp=(Io(pivot,i)-itry(j))/(Io(pivot,i)-Io(pivot-1,i));
```

```
        vtgt=Vo(pivot,i)-((Vo(pivot,i)-Vo(pivot-1,i))*linterp);
        vtotal(j)=vtotal(j)+vtgt;
        maxpwr(j)=maxpwr(j)+(itry(j)*vtgt);
    end
end

itotal=[0 itry min(isc)];
vtotal=[sum(voc) vtotal 0];
[maxp indx]=max(maxpwr);
imaxp=itry(indx);
```

As discussed earlier in this thesis, the routines above resort to linear interpolation to approximate values on an IV curve when they are not specifically known through simulation data. The total error introduced by using this approximation for all four junctions of the cell adds up quickly.

LIST OF REFERENCES

[1] *Solar Cells* Webpage (http://www.solarbotics.net/starting/200202_solar_cells/200202_solar_cells.html), last accessed 23 August 2006.

[2] *Ultra Triple Junction (UTJ) Solar Cells* Data Sheet, Spectrolab, Incorporated, Sylmar, CA.

[3] Science Help Online Chemistry Lesson 3-6 Electron Configuration (http://www.fordhamprep.org/gcurran/sho/sho/lessons/lesson36.htm), accessed 21 Jul 2006.

[4] Silicon electron shell diagram (http://commons.wikimedia.org/wiki/Image:Electron_shell_014_silicon.png) , accessed 21 July 2006.

[5] Physics for Scientists and Engineers, Chapter 42 Quantum Physics Part One (http://www.kineticbooks.com/physics/trialpse/42_Quantum%20Part%20One/toc.html), accessed 21 July 2006.

[6] Michael, S. , *EC3230 Lecture Notes*, Naval Postgraduate School, Winter 2004 (unpublished).

[7] Bates, A., Novel Optimization Techniques for Multijunction Solar Cell Design Using Silvaco ATLAS^{TM}, Master's Thesis, Naval Postgraduate School, 2003.

[8] Man, K.F. , K.S. Tang, and S. Kwong, "Genetic Algorithms: Concepts and Applications",*IEEE Transactions on Industrial Electronics*, Vol. 43, No. 5, pp. 519-534, 1996.

[9] *Atlas User's Manual*. Software version 5.8.3.R, Silvaco International, Sunnyvale, CA, September 2004.

[10] Michalopoulos, P., A Novel Approach for the Development and Optimization of State-of-the-Art Photovoltaic Devices Using Silvaco, Masters Thesis, Naval Postgraduate School, 2002.

[11] Green, M. , The Verification of Silvaco as a Solar Cell Simulation Tool and the Design and Optimization of a Four-Junction Solar Cell, Master's Thesis, Naval Postgraduate School, 2002.

[12] Crespin, A., A Novel Approach to Modeling the Effects of Radiation in Gallium-Arsenide Solar Cells Using Silvaco's ATLAS™ Software, Master's Thesis, Naval Postgraduate School, 2004

[13] Therrien, C. W. and M. Tumamala, *Probability for electrical and computer engineers*. Boca Raton, FL: CRC Press LLC, p. 18,299.

[14] Duncan, R., "A Survey of Parallel Computer Architectures," IEEE Computer. February 1990, pp. 5-16.

[15] Berkeley Open Infrastructure for Network Computing, University of California, Berkeley, http://boinc.berkeley.edu, last accessed September 2006.

[16] Search for Extra Terrestrial Intelligence @ Home, University of California, Berkeley, http://setiathome.berkeley.edu/, last accessed September 2006.

[17] Parallel Virtual Machine, Oak Ridge National Labs, http://www.csm.ornl.gov/pvm/, last accessed July 2006.

[18] LAM/MPI Parallel Computing, http://www.lam-mpi.org/, last accessed July 2006.

[19] Mathworks, Inc, MATLAB™ Distributed Computing Engine Pricing March, North America Academic, March 2006

[20] Jain, R. The Art of Computer Systems Performance Analysis. New York: John Weiley & Sons, 1991.

[21] Utsler, J. and S. Michael, "The Use of Genetic Algorithm for the Design and Optimization of Advanced Multi-Junction Solar Cells", presented at the Midwest Symposium on Circuits and Systems, Cincinnati, August 2005

[22] Utsler, J. and S. Michael, "The Design of Advanced Multi-Junction Solar Cells Using Genetic Algorithm for the Optimization of a Novel Cell Silvaco Model", presented at IEEE World Conference on Photovoltaic Energy Conversion, Honolulu, May 2006

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California

7. Dr. Jeff B. Knorr, Chairman
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

8. Dr. Sherif Michael
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

9. Dr. Bret Michael
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

10. Dr. Todd Weatherford
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California